The Pragmatic Programmers

TURING 图灵程序设计丛书

C++程序设计实践与技巧

测试驱动开发



【美】 Jeff Langr 著

余飞 秦涛 译

- 敏捷大师Bob大叔作序推荐
- 提供大量现代C++编程的实践和技巧
- 深入探讨测试驱动开发、设计原则和优秀软件开发工艺

 中国工信出版集团  人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

Jeff Langr

资深程序员，C++语言专家，曾在Bob大叔的Object Mentor公司工作，后创建Langr Software Solutions公司。出版过多本与测试驱动开发相关的图书，如《Agile Java：测试驱动开发的编程技术》等。

余飞

毕业于吉林大学计算机科学与技术学院。先后就职于Intel和ARM，曾多年从事显卡与3D图形API（OpenGL和Direct3D）驱动程序研发工作。现今从事虚拟现实方面的研发工作。个人的主要专注点为操作系统和3D图形的相关技术，此外对移动应用和云计算技术也颇感兴趣。

秦涛

毕业于浙江大学信息学院控制系。先后就职于AMD、Intel、ARM，曾多年从事显卡驱动研发工作。目前从事虚拟现实方面的研发工作。

图灵程序设计丛书

C++ 程序设计实践与技巧

测试驱动开发

【美】 Jeff Langr 著

余飞 秦涛 译



Modern C++ Programming with Test-Driven Development

Code Better, Sleep Better

人民邮电出版社

北京

图书在版编目（CIP）数据

C++程序设计实践与技巧 ：测试驱动开发 / （美）杰夫·兰格（Jeff Langr）著 ；余飞，秦涛译. -- 北京 ：人民邮电出版社，2017.1
（图灵程序设计丛书）
ISBN 978-7-115-43895-9

Ⅰ. ①C… Ⅱ. ①杰… ②余… ③秦… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2016)第264953号

内 容 提 要

本书是一本关于设计原则、编程实践、测试驱动开发的指南，旨在帮助 C++ 程序员用测试驱动开发方法构建高性能解决方案。全书共 11 章，涵盖测试驱动开发的基本工作方式、潜在好处、怎样利用测试驱动开发解决设计缺陷、测试驱动开发的难点和成本、怎样利用测试驱动开发减少甚至免除调试工作，以及如何长时间维持测试驱动开发。

本书适合所有技术层次的 C++ 程序员阅读。

◆ 著　[美] Jeff Langr
译　余　飞　秦　涛
责任编辑　朱　巍
执行编辑　杨　婷
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：19.25
字数：449千字　2017年1月第1版
印数：1－3 500册　2017年1月北京第1次印刷

著作权合同登记号　图字：01-2014-0500号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

本书赞誉

Jeff Langr又写了一本很棒的书。这一次他把测试驱动开发引入到C++世界。Jeff的示例让我们近距离地领略到好的测试驱动开发方法简约的一面。他解释了为什么要以这种方式工作，然后提出了重要的实践细节，涉及测试替身、与遗留代码打交道的方法、对付多线程代码，还有很多其他的内容。每个使用C++的开发者都应该拥有这本书。

——Ron Jeffries，极限编程方法学创始人之一，《软件开发本资论》作者

Jeff Langr写了近年来最好的一本C++图书。《C++程序设计实践与技巧：测试驱动开发》是理论与实践的完美结合。书中对抽象概念的解释清晰明了、妙趣横生，需要时，细节之处也是唾手可得。本书无疑是C++和测试驱动开发方面的经典之作。

——Michael D. Hill，极限编程方法教练和作家

Jeff Langr是软件开发方面的专家。他在这本书中分享了关于构建优秀软件的智慧。这本书不是讲测试的，但是你依然能学到重要的测试技巧。本书是通过测试驱动开发方法来改善你的技术、代码、产品和生活。无论你是哪个层次的C++程序员，Jeff都将向你展示为什么使用测试驱动开发能构建出更好的C++软件产品，以及如何做到。

——James W. Grenning，《测试驱动的嵌入式C语言开发》作者

译　者　序

回想当初，其实我刚开始打算翻译的是一本关于调试的图书，后因其他缘由翻译了此书。当时想的是，唔，测试驱动开发在平常的开发过程中也会用到，应该不算陌生。看到本书原稿后，我便坚定了翻译此书的决心。一方面，本书就像一本关于测试驱动开发的《一万个为什么》，它回答了一系列的问题，比如什么是测试驱动开发？怎样进行？难点在哪？它为什么能有助于构建好的解决方案？你需要什么样的测试？测试该怎么写？测试需要维护吗？测试驱动开发中所编写的单元测试和非测试驱动开发中所编写的单元测试有何区别？总之，几乎你遇到的疑惑，本书中都有阐述。另一方面，它是以一个个示例的方式来阐述的，代入感非常强。

回头看测试驱动开发本身，它是早期XP（eXtreme Programming）运动的一部分，并且当下的众多敏捷开发团队也大多采用测试驱动开发或其变种。测试驱动开发的增量开发和敏捷流程相契合，能够提供更快的反馈，而快速的反馈意味着开发将更加精准和灵活。另外，测试驱动开发中所编写的测试会让你在不断调整代码的过程中不轻易破坏已有的行为。敏捷专家Bob大叔曾声称他的Fitness有将近6.4万行代码，而测试占据了2.8万行。这些测试可在90秒内运行一遍。后来Fitness中又加入了2万行代码，而整体代码的缺陷仅为10多个，这些测试显然为代码的持续修改和快速发布夯实了基础。

测试驱动开发的固有周期可以轻松地将你带入一个有序的开发轨道。在每个周期中，你将专注于思考真正的需求、解决方案及设计重构。它将你置身于一个框架下，不断锤炼你在各个周期所需要的技能。在测试的编写阶段，你会更加细致地思考需求，考察代码覆盖率等；在编写产品代码阶段，你可以专注于初始架构设计和编程实践；在代码重构期，除了重新考量架构，你还可以做任何你想做的优化。当然，有序的开发轨道并不意味着一路顺畅，你依然会在每个时期遇到各类挑战，这时你需要根据形势作出正确的判断和决定。

刚开始接触测试驱动开发的开发者可能会觉得用这种开发方式节奏有点慢。可能慢在写测试（下一个测试是什么，测试的命名，怎么写测试），也可能慢在审阅和重构。一开始，你花在测试上的时间可能要多于开发产品代码的时间，但坚持住，你所作出的努力都会得到回报的。不知道读者朋友们有没有过练习一门武术的经历。译者有幸曾跟随过一位旅居上海的华人练习咏春直至他离开上海，师傅经常提醒我，有的动作要慢慢练，慢到别人几乎看不出来你在动。所谓天下武功，唯快不破！这个快其实是无数慢速训练量变后所呈现的质变。在修习测试驱动开发时，有时候你也要慢慢做。在这个慢的过程中你一定会看到以前没有看到过的风景。

测试驱动开发易学难精，其原因在于各类软件或其模块的行为复杂度不一。此外，测试驱动开发的内容繁杂，在总体呈优的情况下也存有一些缺点。在运用测试驱动开发时，必要时需结合所开发软件的特点，做到扬弃地使用。最后，测试驱动开发符合事物发展的一般规律，其自身也在不断发展，本书只能算是引领你登堂入室的第一课，所以不要止步于此！要不断学习、实践和总结。当然，除了技术上的因素，处于乐于推行测试驱动开发的工作环境也是十分必要的。坦白讲，我第一次在生产环境下试图使用真正的测试驱动开发时，过程还是挺狼狈的。

翻译是一个学习、积淀的过程，当然也是艰辛的。感谢我的家人，特别是我的妻子从一开始就非常支持我，翻译的过程占据了大量本可以陪伴家人的时间，但是你们却一如既往地给我支持。

在翻译本书的后期，由于工作和生活上的事情使得翻译时间缩减，因此我邀请了好友秦涛帮忙翻译第9章和第10章。多谢你在繁忙之中拔冗相助，使得翻译得以及时完成。

感谢朱巍老师从一开始就给予的信任和支持。感谢各位编辑老师为本书付出的辛勤汗水。没有你们的帮助和支持，很难想象出版本书是怎样的过程。

由于时间和水平有限，文中肯定存有一些瑕疵，或者读者朋友们有什么问题，都可以发邮件至fei_faith@outlook.com作进一步讨论。

最后以Langr为原书提供的副标题与各位程序设计同仁共勉：Code Better, Sleep Better！

余飞

2016/10/10，上海

序

不要被书名误导。

我的意思是，这是一本关于设计原则、编码实践、测试驱动开发和技艺的非常非常好的书，却起了个“Modern C++ Programming with Test-Driven Development:Code Better,Sleep Better”的名字。唉！

不要误会，这本书的确是关于现代C++编程的。如果你是C++开发者，你会喜欢上书中所有的代码。这本书中充满了有趣的、编写良好的C++代码。事实上，我认为代码略多于文字描述。翻阅一下这本书吧。你能找到没有代码的一页吗？我敢打赌没多少！所以，如果你正在找一本现代C++实践方面配以大量示例的好书，那么你算是找对了！

但是，这本书讨论的内容远不止现代C++编程，而是讲了很多很多！

首先，这是我见过测试驱动开发方面内容最完整、论述最好的书（我看过的书太多了）。几乎在过去十五年里，未经讲述的每个测试驱动开发问题在这本书中都有讨论。从脆弱性测试到模拟（mock），从伦敦流派[1]到Cleveland流派，从Single Assert到Given-When-Then[2]，这本书中都讨论了，而且还不止这些。此外，这本书并非泛泛地概述一些没有关联的问题。相反，它详述每个问题，配以示例和讨论。可谓是问题现于代码，也解于代码。

需要先成为C++程序员才能理解这本书吗？当然不需要。书中的C++代码十分整洁，概念也十分清晰，所以，Java、C#、C甚至Ruby程序员理解起来都没问题。

其次，书中讲述了设计原则！谢天谢地，这本书是一本设计教程！它带你遍及令人目不暇接的原理、问题和技巧。从单一功能原则到依赖倒置原则，从接口隔离原则到简洁设计背后的敏捷开发原则，从DRY[3]到Tell-Don’t-Ask[4]。本书收纳了各种软件设计方法和方案，而且这些方法就

① 测试驱动开发中有多种流派，其中伦敦流派起源于创立于伦敦的ETC（Extreme Tuesday Club），这个俱乐部提倡敏捷开发。——译者注

② Give-When-Then是一种指导编写测试的模板。可以参考http://guide.agilealliance.org/guide/gwt.html。——译者注

③ DRY是Don’t Repeat Yourself的缩写，此原则倡导消除各式各样的信息冗余。更多内容请参考https://en.wikipedia.org/wiki/Don%27t_repeat_yourself。——译者注

④ 这个原则倡导一个对象应该命令其他对象该做什么，而不是去查询其他对象的状态来决定做什么。遵循此原则会带来更好的封装性。——译者注

呈现在真实问题和真实代码解决方案之中。

之后，书中还讲述了编码实践和技巧。这本书通篇都在讲解这些内容，从小方法到结对编程，从编程招式到变量名。书中不仅有大量的代码供你一窥优秀的实践和技巧，作者还以恰当的讨论和演绎入木三分地解释了这些主题。

是的，书名是彻底的败笔。这不是一本讲C++的书，而是一本论述优秀软件开发工艺的书，只是恰巧使用C++来书写示例罢了。真的，书名应该叫“软件技艺：现代C++示例”（Software Craftsmanship: With Examples in Modern C++）。

所以，如果你是一名Java程序员，或者C#程序员，抑或是Ruby、Python、PHP、VB甚至COBOL程序员，都会有阅读此书的冲动。不要被封面上的C++字样吓到。不管怎样，先读一读。阅读的时候，也读一读附带的代码。你会发现它很容易理解。在学习好的设计原则、编码技巧、技艺和测试驱动开发时，你或许会发现学一点点C++其实也无妨。

“Bob大叔”Martin

Object Mentor公司创始人

前　言

尽管当下程序设计语言呈现爆炸式发展，C++却一如既往地坚挺。在2013年7月Tiobe最受欢迎的编程语言排行榜上，C++位居第四（最新的排名可以参见http://www.tiobe.com/index.php/content/paper-info/tpci/index.html）。2011版ISO标准（ISO/IEC 14822:2011，即C++11）定义了一些新的C++特性，这些特性可以提升C++的接受度，至少能减弱反对使用C++的声音。

对于构建高性能解决方案，C++依然是最好的选择之一。如果你所在公司的产品要与硬件集成，那么很有可能公司已经拥有了一个基于C++的庞大系统。如果你的公司在20世纪90年代或更早之前就存在，很有可能拥有一个使用了很长时间的基于C++的系统，而且它在未来的几年内也不会消失。

假设在工作中使用C++，你可能会思考以下几件事情。

- 现在是2013年，为什么我会回头使用这一难用（但好玩）并且早在几年前就想摈弃的编程语言？我该怎样做才不会自讨苦吃？
- 我是一个C++老手，对这门功能强大的语言了如指掌。 多年来我一直使用它顺利地进行开发工作。我为什么要改变工作方式呢？
- 我的职业前景在哪里？

谈谈我自己吧！我从20世纪90年代初期开始使用C++，那个时候还没有模板（和模板元编程）、RTTI、STL和Boost这样的东西。从那时起，我有几次必须使用这门强大的语言，但是结果有点令人沮丧。和众多编程语言一样，C++也时常会让你搬起石头砸自己的脚，但使用C++的不同之处是，有些时候你不会意识到即将要砸到脚，等真正发生时已为时过晚。与使用其他编程语言相比，你可能会经受更多的痛苦。

如果你已经使用C++工作了很多年，可能采用了许多惯用法来提高代码的质量。在所有编程语言开发者中，铁杆的C++老手相对更加仔细，因为长时间使用C++开发而不出问题，需要对代码给予一丝不苟的关注和照料。

你或许会想，如此精心照料，C++系统的代码质量应该很高吧！但是，大部分C++系统都会暴露出如下一些相同的问题，而且会不断地出现。通常，这些问题和编程语言无关：

- 几千行的巨量源代码；

- 包含成百上千行晦涩代码的成员函数；
- 大量的无用代码；
- 编译时间超过几个小时；
- 居高不下的代码缺陷数目；
- 快速修复导致逻辑复杂晦涩而难以安全地管理；
- 散落在多个文件、类和模块间的重复代码；
- 使用早已废弃的编程实践。

这些问题不可避免吗？答案是可以避免，测试驱动开发（Test-Driven Development，TDD）正是这样的工具，你可以掌握并运用它来降低系统的熵[①]。它甚至可以重振你对编程的热情。

倘若你只是想找份工作，大量的C++工作机会能让你不愁就业问题。但是，C++是一门高度讲究技巧、很微妙的编程语言，粗心大意地使用会导致代码缺陷、间歇性故障和旷日持久的调试工作。这些也会使你的工作没有保障。好在测试驱动开发帮得上忙。

往一个庞大且存在了很长时间的C++系统中加入新的功能通常很耗时，而且无法估计进度。只是理解一段代码然后修改几行，有可能要花费几个小时甚至几天。开发者需要花上数小时等待代码改动编译完成，并且要等待更长的时间去检测这些改动是否和现有的系统完好集成，这会导致生产力进一步下降。

其实这都是可以避免的。早在20世纪90年代末，TDD就被用来帮助不断往系统中加入新的特性，同时保持C++系统在可控的范围内。（先于代码写测试的观念已经存在很长一段时间了，但是，正式的TDD周期是Ward Cunningham和Kent Beck在《测试驱动开发》一书中提出的。）

本书的主要目的是教你实际使用TDD的系统方法。你将会学到：

- TDD的基本工作方式；
- TDD的潜在好处；
- TDD怎样帮助解决设计缺陷；
- TDD的难点和成本；
- TDD怎样减少甚至免除调试工作；
- 怎样长时间维持TDD。

它适合我和我的系统吗

“这和单元测试有什么关系？似乎并不能帮到我太多。”

或许你已经尝试过单元测试，抑或正在努力为一个旧系统编写单元测试。TDD似乎对于工作

① 熵是热力学的概念，是在物质微观热运动时，用以表达其混乱程度的量。最初由香农引入信息论中，用以衡量信息的不确定性或随机性。这里亦可以表示软件系统混乱、不稳定的程度。——译者注

在新系统上的幸运儿是好东西，但它能解决C++系统上长期存在的棘手的日常问题吗?

的确，TDD是一个有用的工具，但不是对付旧系统的良方。虽然可以在开发系统新特性时使用TDD，但依然需要清理系统中多年积累的障碍。为了持续这样的清理工作，需要额外的战略和战术。你需要学习战术性的依赖消除法和安全的代码修改方法，这些可以在Michael Feathers的《修改代码的艺术》一书中找到。你还需要理解在不引起很多问题的情况下，怎样进行大规模的重构。这方面的内容可以参考*Mikado Method* [BE12]。本书也会传授此类实践和其他技术。

通常，只为努力阐明“系统就是这样的”而为现有代码添加单元测试[我称之为开发后测试（Test-After Development，TAD）]，收效甚微。可能投入了大量人力编写此类测试，但对系统的质量影响却很小。

若要让TDD帮助塑造系统，你的设计要从定义上是可测试的。这么做会有所不同，在许多方面要好于未采用TDD。越懂得什么是好的设计，TDD就越能帮你达到此目的。[①]

为了协助你转变思考设计的方式，本书强调潜藏于优秀代码管理方法背后的原则，如面向对象设计中的SOLID原则，这在《敏捷软件开发：原则、模式与实践》一书中有所讲述。本书将讨论优秀的设计理念怎样保持持续的开发和生产力，还有TDD怎样帮助有心的开发者达到更一致、可靠的结果。

本书目标读者

本书旨在帮助所有技术层次的C++程序员，无论是对C++语言有基本理解的新手，还是长期浸淫在语言秘籍中的老手。如果已经有一段时间没用C++了，那么你会发现TDD的快速反馈周期会帮助你快速地重新拾起这门编程语言。

尽管本书的目的是传授TDD，但是无论使用TDD的经验是多还是少，你都会有所收获。如果你完全对编写单元测试没有概念的话，我会带着你逐步地了解TDD的基础知识。如果你是第一次听说TDD，会在本书中发现很多专家建议，这些建议以简单的方式呈现，配以直截了当的例子。甚至经验老到的测试驱动开发者也能找到一些有用的智慧结晶、有关实践的可靠理论基础，以及一些有待探索的新主题。

如果你是怀疑论者，则可以从多个视角探索TDD。我会在整本书中解释为什么我认为TDD能够很好地工作，我也会分享一些它不能很好工作的经历以及原因。本书不是一本宣传手册，而是革命性技术的探索旅程，令人大开眼界。

各类读者也会找到在团队中发展和维持TDD的办法。TDD很容易上手，但是你的团队可能会遇到很多挑战。怎样避免你的变革不被这些挑战折损？怎样避免灾难？第11章会提供一些我认为行之有效的方法。

---

① 一个好的设计，其内部行为规整有序。TDD有助于保持这种好的设计行为不变性。——译者注

阅读前提

为了使用随书附带的所有例子，你需要一个编译器和一个单元测试工具。有些例子会用到第三方程序库。下面将概览这三种要素。可以参见第1章进一步了解细节。

单元测试工具

在众多的C++单元测试工具中，我选择Google Mock（基于Google Test）作为书中大部分例子的工具。目前来说，它是关注度最高的，但是我选择它的主要原因是，它支持Hamcrest表示法[一种基于匹配器（matcher）的断言形式，用以提供具备较强表达力的测试]。第1章提供的信息将帮助你快速使用Google Mock。

但是，本书既不是Google Mock的专著，也不是其宣传手册，而是教授TDD的。你只会学到足够的Google Mock知识，用以有效地实践TDD。

对于有些示例，还需要另外一个单元测试工具——CppUTest。如果没有使用过Google Mock或者CppUTest，也大可放心，因为你会发现学习另外一个单元测试工具是非常容易的。

倘若你正在使用不同的单元测试工具，诸如CppUnit或者Boost.Test，也不用担心！这些工具和Google Mock在概念上很像，在实现方式上也类似。你可以轻松跟上，并且几乎可以用任何其他的C++单元测试工具来做TDD的例子。可以参见附录A，了解在选择单元测试工具时什么是最重要的。

本书中的大部分例子使用Google Mock用以mocking和stubbing，参见第5章。当然，Google Mock和Google Test是一起工作的，但是也能将Google Mock成功地集成进你所选的单元测试工具。

编译器

需要一个支持C++11标准的编译器。本书中的例子最初是使用gcc编译的，在Linux和Mac OS上可以直接使用。参见第1章获取如何在Windows上编译的信息。所有的例子都使用了STL，它是许多平台上现代C++开发中的基本组成部分。

第三方程序库

一些例子使用了免费的第三方程序库。参见第1章获取需要下载的程序库列表。

怎样使用本书

我在设计本书章节时尽量使它们功能独立，你可以随机挑选一章阅读，而不需要依赖其他章

节。如果使用电子阅读器的话，我也提供了充足的全文交叉索引，可方便、轻松地在章节间跳转。每一章都有一个总览和一个总结，并配以下一章预览。这些概要性小节的名称，与许多单元测试框架的初始化代码段和清理代码段一样，取名为Setup和Teardown[①]。

源代码

本书包含大量的代码示例。大部分代码与一个特定的文件名联系。你可以在本书的网站[②]上找到所有代码：http://pragprog.com/book/lotdd/modern-c-programming-with-test-driven-development。或者访问我的GitHub主页：http://gitbug.com/jlangr。

示例代码按照章节组织。在每一章的目录下，有很多以数字命名的目录，每个数字对应一个版本号。在各章的学习过程中，我们会引用这些版本号说明代码的演变。举个例子，以c2/7/SoundexTest.cpp为标题的代码，在第2章代码目录下第7个版本目录的SoundexTest.cpp文件中。

本书讨论

请加入本书论坛：https://groups.google.com/forum/?fromgroups#!forum/modern-cpp-with-tdd。论坛的目的是讨论书中内容，以及和C++测试驱动开发有关的方面。我也会发布一些和本书相关的有用信息。

如果你第一次接触测试驱动开发：本书包含什么

虽然本书适合所有人，但其主要受众是第一次接触TDD的程序员，所以书中的章节安排有相应的先后顺序。我强烈建议你完成第2章的练习。完成这样一个示例，你会对TDD背后的思想有深刻的领悟。切忌只是阅读，要尽量动手去做，并保证该通过的测试都能通过。

接下来的第3章和第4章是必读的。这两章详述了什么是TDD（和什么不是TDD）以及怎样构建测试。在学习mock之前（参见第5章），务必保证理解这两章的内容。mock是构建大多数产品级系统的必备技术。

不要自以为了解设计和重构而跳过第6章。践行TDD的重要原因是，它使你在通过持续重构改进设计的同时，能够保持代码干净整洁。大部分系统有着糟糕的设计和晦涩的代码，部分原因在于，开发者不愿意做足够的重构或者不知道怎么去做。你将学到足够的知识，了解怎样获益于一个更小、更简单的系统。

---

① 一般而言，每个测试会提供Setup和Teardown函数，用于测试的初始化和清理操作。作者特意为书中每章配备这一头一尾两小节，与测试框架的结构相呼应，颇具创意。相比于测试代码，为了便于书面阅读，Setup和Teardown在文中分别译作“开场白”和“结束语”。——译者注

② 本书在图灵社区上的地址是：http://www.ituring.com.cn/book/1303。——编者注

第7章将总结TDD的核心技巧。这一章考察了许多方法，它们能提高你在TDD上投资的回报。学习其中的一些技巧将能够决定TDD成功与否。

你肯定会被卷入一场与已有系统的斗争，而这个已有系统可能没有使用TDD。可以阅读第8章了解一些应对遗留代码的简单技术。

第9章专门讨论多线程下的TDD。TDD方法或许会让你眼前一亮。

第10章将深入一些特定的领域和关注点。你会发现一些TDD的最新思路，包括与本书不同的一些方法。

最后，你想知道怎样使你的团队采用TDD。当然，你也一定想确保持续使用TDD。第11章会提供一些你想要收入囊中的点子。

如果你有一些测试驱动开发经验

你可以随机阅读各个章节，但也会发现许多来之不易的智慧之言贯穿全书。

排版约定

书中会穿插一些长度不一的代码段。当正文引用这些代码时，将使用下面的约定。

- 类名和测试名采用大驼峰式命名法（UpperCamelCase）。
- 其他的代码内容以代码字体呈现。下面是一些例子：
  - 函数名（显示空参数表，即便所指的函数有一个或多个参数。有时候，我称成员函数为方法）;
  - 变量名；
  - 关键字；
  - 其他代码片段。

为了保持简洁并节省版面，代码清单通常会省略与当前讨论不相干的代码。大括号后的注释代替了省略掉的代码。如下所示，for循环体被省略。

```
for (int i = 0; i < count; i++) {
    // …
}
```

关于“我们”

本书是我们之间进行交流的载体。一般而言，我是在与读者谈话。当我特指“我”的时候（不是很经常），通常是在表述经验之谈或个人偏好。隐含之意是，这些表述可能并没有被广泛接受

（虽然有可能是个好想法）。

具体到编码，我希望不是你一个人在做，原因在于你仍在努力学习。我们将共同完成书中的编码练习。

关于我

我从1980年就开始编写程序了，当时我在读高二。我的编程职业生涯是从1982年开始的，当时我在马里兰大学工作，同时也在攻读计算机科学学士学位。2000年，我完成了从程序员到咨询顾问的角色转变，期间我很享受为Bob Martin工作，以及偶尔与Object Mentor的优秀人才一起工作的时光。

2003年，我创办了Langr Software Solutions，提供敏捷开发相关的咨询和培训服务。我的大部分工作是与开发者一起实践TDD，或者教授TDD。在真实的开发团队中，我坚持在咨询师/培训师和一线程序员两种角色间来回切换，以保证自己一直参与其中不掉队。从2002年起，我作为全职程序员为四个不同的公司工作了相当长一段时间。

我喜欢从事软件开发方面的写作。这是我深入学习事物的一种方式，我也乐于帮助其他开发者快速编写出高质量代码。这是我的第四本书，其他三本是*Essential Java Style: Patterns for Implementation* [Lan99]、*Agile Java: Crafting Code With Test-Driven Development* [Lan05]和*Agile in a Flash* [OL11]，其中最后一本是与Tim Ottinger合著的。我为Bob大叔的《代码整洁之道》撰写了几章。除了我自己的网络站点，我还在其他站点上写过一百多篇文章。我会定期更新自己的博客（http://langrsoft.com/jeff），也为Agile in a Flash项目（http://agileinaflash.com）写过或参与写作上百篇文章。

除了C++，我们还使用过许多其他编程语言，如Java、Smalltalk、C、C#和Pascal等。目前，我在学习Erlang，也能够使用Python和Ruby进行编程。除此之外，我学习过至少十几种编程语言来了解它们（有时候也是为了一时之需）。

本书中的 C++代码风格

虽然我使用C++开发过各种规模的软件系统，但我并不认为我是一个语言专家。我读过Meyers和Sutter的一些重要著作，当然还有其他一些。我知道怎样让C++代码工作，以及如何使代码更具表达力和可维护性。我知晓这种语言的大部分深奥的用法，却尽量避免使用它们。我在本书中对聪明代码的定义是“难以掌控”。我会引导你另辟蹊径。

我写C++代码的风格偏向于面向对象（这当然是受到Smalltalk、Java和C#的影响）。我喜欢大部分代码以类的方式存在。本书中大部分代码遵从这一风格。例如第一个示例中的Soundex就是以类的方式构建的（参见第2章），当然不是必须这样做。我喜欢这种方式，如果你不喜欢，可以

用自己的方式来实现。

TDD的价值本身与C++编程风格无关，所以不要受我的风格影响而否定其潜力。过多强调面向对象有助于对测试替身（参见第5章）的理解，这个时候需要拆除难缠的依赖关系。如果置身于TDD中，时间久了或许会让你的编码风格向面向对象转变。这倒不是坏事！

我有点懒。鉴于示例规模较小，我尽量不用命名空间，但在实际的产品代码中我一定会使用它。

我喜欢尽量保持代码精简，这样会避免我认为的视觉混乱。因此在大部分实现文件中，你会发现use namespace std;，尽管许多人认为这是不好的风格。（保持类和函数小巧且功能单一，比"所有的函数应该有唯一的返回值"这样的指南更有用。）不必担心，TDD不会影响你坚持自己的标准，我也不会。

最后关于C++要说的是，这门语言博大精深。我确信有更好的方法来实现本书中的示例，而且我敢肯定有一些我没使用过的库。测试驱动的优点是，你可以重新以多种方式完成一个实现，却不用担心损害其他功能。无论如何，请写信告诉我一些改善的建议，前提是你愿意使用TDD的方式。

电子书

扫描如下二维码，即可购买本书电子版。



致　　谢

感谢我的编辑Michael Swaine和PragPorg的伙伴们，他们提供了指导和书中所用的资源。

感谢Uncle Bob为本书精彩作序！

非常感谢我的技术编辑Dale Stewart，他在整个成书过程中提供了非常有价值的帮助，尤其是对书中示例的反馈和帮助。

在写作过程中，我总是寻求真诚的反馈，而Bas Vodde针对全书提供了大量反馈。他是一个真诚而有力的幕后搭档。

特别感谢Joe Miller，他不辞劳苦地将示例移植到Windows下。

非常感谢提供想法和重要反馈的人们：Steve Andrews、Kevin Brothaler、Marshall Clow、Chris Freeman、George Dinwiddie、James Grenning、Michael Hill、Jeff Hoffman、Ron Jeffries、Neil Johnson、Chisun Joung、Dale Keener、Bob Koss、Robert C. Martin、Paul Nelson、Ken Oden、Tim Ottinger、Dave Rooney、Tan Yeong Sheng、Peter Sommerlad和Zhanyong Wan。如果漏掉了哪位，请接受我的道歉。

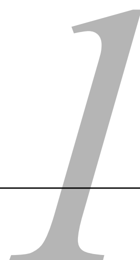
感谢对PragProg勘误页面提供反馈的人们：Bradford Baker、Jim Barnett、Travis Beatty、Kevin Brown、Brett DiFrischia、Jared Grubb、David Pol、Bo Rydberg、Jon Seidel、Marton Suranyi、Curtis Zimmerman，等等。

再次感谢Tim Ottinger，他撰写了前言的一部分，也为本书出谋划策。我怀念与你共事的日子！

谢谢你们帮助我让本书变得更好，只靠我自己是做不到的。

目　　录

第1章 环境设置



1.1 开场白

搭建开发环境是任何软件开发过程中都需要劳心费力的事情之一。在本章中，你将了解到编译和执行书中示例所需的工具。你也会了解一些相关的提示，以避开我走过的弯路。

本章目前包含Linux和Mac OS下的环境搭建。如果是在Windows下使用C++的话，可以参考1.3.3节提到的建议。

1.2 示例程序

书中示例的源代码可以从http://pragprog.com/titles/lotdd/source_code下载。每章的示例放在一组。

从TDD中学习的许多东西都涉及增量地开发代码，所以每章的示例都以增量的方式给出。每章的目录下以数字命名的子目录（1、2、3……）对应相应的版本。例如，第2章中的第一段代码是c2/1/SoundexTest.cpp，它表示SoundexTest.cpp的第一个版本。c2/2/SoundexTest.cpp则是第二个版本。

也能在GitHub（https://github.com/jlangr）上找到示例程序。在GitHub上，可以找到对应本书相关章节的代码库。例如，代码库c2对应本书第2章的Soundex示例。

本书中示例的代码版本号对应GitHub版本库的一个分支（branch）①。例如，可以在代码库c5的分支4找到c5/4/PlaceDescriptionService.cpp所对应的代码。

在每个版本的目录下，可以找到源代码，其中有一个主函数来运行所有的测试，还有一个CMake编译脚本。需要安装和配置一些工具来运行示例程序。有一些示例需要安装额外的第三方库。

① 对于没有接触过GitHub的开发者，可以先了解一下Git（参见http://www.git-scm.com），它是一个分布式的版本管理工具。GitHub提供的代码管理正是基于Git。当然，作者只是提供一个从GitHub上下载示例的选项，理解本书的内容并不需要懂得Git。——译者注

你需要一个支持C++11标准的编译器和make[1]来编译示例代码。大部分代码需要Google Mock作为单元测试工具。其中有三章的示例使用CppUTest作为单元测试工具。

你可能需要修改源码包来支持其他的编译器（或者不支持C++11的编译器），使用其他的编译工具或单元测试工具。所幸的是，除了第7章的库代码，大部分示例都比较小。

下面的表格列举了每章对应的目录，以及所需的单元测试工具和第三方库。

章节名称	目录	单元测试工具	第三方库
测试驱动开发：第一个示例	c2	Google Mock	无
测试驱动开发基础	c3	Google Mock	无
测试结构	c4	Google Mock	无
测试替身	c5	Google Mock	cURL、JsonCpp
增量设计	c6	Google Mock	Boost（格列高利历）
高质量测试	c7	Google Mock	Boost（格列高利历、算法、赋值）
遗留代码的挑战	wav	CppUTest	rlog、Boost（文件系统）
测试驱动开发与多线程	c9	CppUTest	无
测试驱动开发的其他概念和讨论	tpp	CppUTest	无
代码Kata：罗马数字转换器	roman	Google Mock	无

1.3　C++编译器

1.3.1　Ubuntu

最初在Ubuntu 12.10上使用g++4.7.2构建书中的示例。

可以通过下面的命令安装g++：

```
sudo apt-get install build-essential
```

1.3.2　OS X

我在Mac OS X 10.8.3（Mountain Lion）上用gcc port成功构建了书中的示例。在撰写本书的时候，随Xcode发布的gcc版本是4.2，这个版本并不能构建书中的示例。

为了安装gcc port，需要安装MacPorts，它允许你在Mac上安装免费软件。参见http://www.macports.org/install.php获得更多的信息。

安装好后，需要使用以下命令更新MacPorts：

```
sudo port selfupdate
```

[1] make是用来完成自动编译的工具，参见https://en.wikipedia.org/wiki/GNU_make。

使用下面的命令安装gcc port（安装可能需要较长的时间）：

```
sudo port install gcc47
```

（如果你喜欢，可以在命令后追加+uniersal选项，这样就可以同时为PowerPC和Intel架构生成程序。）

在成功安装好gcc port后，需要指定其为默认的gcc。可以使用下面的命令：

```
sudo port select gcc mp-gcc47
```

可以使用下面的命令将gcc加入路径名称列表：

```
hash gcc
```

1.3.3 Windows

在Windows操作系统上，要保证代码的行为和预期一样，可以考虑使用MinGW或Cygwin中的g++ port。你可以尝试Microsoft Visual C++编译器2012年11月的CTP版和Clang，但是截至本书写作时，它们都还不能完整地支持C++11标准。本节将简要描述一下在Windows下运行本书示例的一些困难和建议。

1. Visual C++编译器2012年11月CTP版

可以下载Visual C++11编译器的社区科技预览（CTP）版[①]。Visual C++团队的博客[②]上有一篇描述相关版本的文章。

快速地看一下开发本书示例使用的CTP版可以很快地了解如下几件事。

- 目前还没完全支持类内成员初始化。
- std库对于C++11的支持似乎是最大的短板。例如，集合类目前还不支持统一初始化列表。std::unordered_map也没实现。
- Google Mock/Google Test使用了可变参数模板，这还没得到完全支持。在构建Google Mock时，你会收到编译错误信息。你需要为这些工程项目加一个预处理宏定义_VARIADIC_MAX，并将其设置为10。可以参见http://stackoverflow.com/questions/12558327/google-test-in-visual-studio-2012获取更多的信息。

2. Windows示例代码

在本书即将付梓之际，我们正在创建能够在Windows上运行的示例（通过移除一些不支持的

① http://www.microsoft.com/en-us/download/details.aspx?id=35515（本书写于2012年，作者当时使用的Visual C++比较旧。最新的Visual Studio 2015完全支持C++11标准。为了得到最新的Visual Studio，读者可以访问https://www.visualstudio.com/。原书给出的Visual C++编译器2012 CTP链接因意义不大，故此处略去。——译者注）

② http://blogs.msdn.com/b/vcblog/archive/2012/11/02/visual-c-c-11-and-the-future-of-c.aspx

C++11特性）。可以在一个单独的GitHub页面（https://github.com/jlangr）上找到修改过的示例，这也是一章一个代码库。可以访问Google讨论组（https://groups.google.com/forum/?fromgroups#!forum/modern-cpp-with-tdd）获得更多Windows示例的信息。

Windows版的代码库包含解决方案（.sln）和项目（.vcxproj）文件。可以使用这些文件在Visual Studio Express 2012中加载示例代码，也可以在命令行中使用MSBuild编译和运行示例。

如果要自己修改代码，应该不是那么困难。去掉类外的初始化应该比较容易。可以使用std::map取代std::unordered_map。许多加入到C++11中的特性来自boost::tr1库，所以你可以直接替换掉Boost库的实现。

3. 一些Windows小技巧

对于编译警告、错误和一些其他导致不能编译的问题，我在网上进行了相关搜索。下面是我学到的一些东西。

错误/问题	解决方案
C297: 'std:tuple': too many template arguments.	添加预处理器定义_VARIADIC_MAX=10。参考http://stackoverflow.com/ questions/8274588/c2977-stdtuple-too-many-template-arguments-msvc11
Specified platform toolset (v110) is not installed or invalid.	设置VisualStudioVersion为11.0
msbuild.exe在哪里	我机器上的msbuild.exe是在 c:\Windows\Microsoft.NET\Framework\v4.0.30319里
Warning C4996: 'std::_Copy_impl': Function call with parameters that may be unsafe.	-D_SCL_SECURE_NO_WARNINGS
按Ctrl-F5运行测试结束后，控制台窗口关闭	设置 Configuration Properties → Linker → System → SubSystem 为 Console（/SUBSYSTEM:CONSOLE）
Visual Studio尝试链接仅存在于头文件中的Boost功能	添加BOOST_ALL_NO_LIB预处理器

许多问题的解决方案已经包含在项目文件中了。

4. Visual Studio 2013预览版

就在本书最后修订的时候，微软发布了Visual Studio 2013预览版，它提供了C++11标准特性的进一步支持，同时还支持了C++14预案特性。短期内，GitHub上的Windows代码可以在2012年11月的CTP版上运行。我们（我自己和一些提供帮助的人）已经使用了Visual Studio 2013，过不了多久，你就能下载到使用了更多C++11特性的升级版本。我希望最终不需要为Windows提供专用版本的示例代码。这是一个符合C++11标准的Windows版编译器！

1.4　CMake

无论好坏，我选择CMake来支持跨平台编译。

对于Ubuntu用户，本书使用CMake 2.8.9编译示例。可以通过下面的命令安装CMake：

```
sudo apt-get install cmake
```

对于OS X用户，本书使用CMake 2.8.10.2编译示例。可以访问http://www.cmake.org/cmake/resources/software.html，下载安装CMake。

当使用CMake来运行编译脚本时，可能会遇到下面的错误:

```
Make Error: your CXX compiler: "CMAKE_CXX_COMPILER-NOTFOUND" was not found.
Please set CMAKE_CXX_COMPILER to a valid compiler path or name.
```

这个信息提示没有找到合适的编译器。如果只是安装了gcc而不是g++，就可能遇到这样的错误。在Ubuntu上，安装build-essential可以解决这个问题。在OS X上，定义或者修改CXX可以解决这个问题。

```
export CC=/opt/local/bin/x86_64-apple-darwin12-gcc-4.7.2
export CXX=/opt/local/bin/x86_64-apple-darwin-12-g++-mp-4.7
```

1.5 Google Mock

本书许多示例都使用了Google Mock，它是一种模拟（mock）和匹配器框架，其中包含了单元测试工具Google Test。我在书中会交替使用这两个术语，但是大部分时候会使用Google Mock以保持简单。你可能需要阅读Google Test文档来理解我提到的Google Mock特性。

需要把Google Mock链接到示例中，这意味着必须先编译Google Mock库。下面的步骤或许能帮到你。也可以参考Google Mock附带的README.txt文件来了解更详细的安装步骤：https://code.google.com/p/googlemock/source/browse/trunk/README。

1.5.1 安装 Google Mock

Google Mock的官方网站是：https://code/google.com/p/googlemock/。可以从http://code/google.com/p/googlemock/downloads/list下载Google Mock。本书示例使用的是Google Mock 1.6.0。

解压安装包（例如，gmock-1.6.0.zip），或许在home目录下。

创建环境变量GMOCK_HOME指向该目录，例如：

```
export GMOCK_HOME=/home/jeff/gmock-1.6.0
```

Windows系统上使用下面的命令：

```
setx GMOCK_HOME c:\Users\jlangr\gmock-1.6.0
```

1. Unix

对于Unix用户，如果你想跳过README里的编译步骤，也可以使用下面的步骤成功编译。

我选择CMake来编译Google Mock。在Google Mock安装根目录下（即$GMOCK_HOME)，执行下列步骤：

```
mkdir mybuild
cd mybuild
cmake ..
make
```

编译目录的名字为mybuild，这是随意的。但是本书示例的编译脚本都使用这个名字。如果使用其他名字，需要改动所有的CMakeLists.txt文件。

你也需要编译Google Test，它包含在Google Mock里。

```
cd $GMOCK_HOME/gtest
mkdir mybuild
cd mybuild
cmake ..
make
```

2. Windows

在Google Mock发布包里，可以找到.\msvc\2010\gmock.sln文件，这个应该适用于Visual Studio 2010和更新的版本。（也可以找到.\msvc\2005.gmock.sln，它适用于Visual Studio 2005和Visual Studio 2008。）

为了能使用Visual Studio 2010和Visual Studio 2012编译Google Mock，需要配置项目使用2012年11月的CTP版。从项目属性开始，导航至Configuration Properties->General->Platform Toolset，然后选择CTP。

CTP版不支持可变参数模板（Visual Studio 2013或许可以）。所以我们来模拟它。[①]需要添加一个预处理定义_VARIADIC_MAX，使其高于默认值5。定义成10应该可行。

使用Google Mock创建项目时，需要指定正确的头文件，包含目录和库文件。导航至Configuration Properties->VC++ Directories，做下面几件事情。

- 添加$(GMOCK_HOME)\msvc\2010\Debug到Library Directories。
- 添加$(GMOCK_HOME)\include到Include Directories。
- 添加($GMOCK_HOME)\gtest\include到Include Directores。

导航至Linker->Input，添加gmock.lib到Additional Dependencies。

要确保Google Mock和项目使用相同的内存模型。Google Mock默认使用/MTd。

① http://stackoverflow.com/questions/12558327/google-test-in-visual-studio-2012

1.5.2 创建 main 函数运行 Google Mock 测试

本书每个示例的代码都有一个main.cpp文件，专门与Google Mock配合使用。

c2/1/main.cpp

```
#include "gmock/gmock.h"

int main(int argc, char** argv) {
    testing::InitGoogleMock(&argc, argv);
    return RUN_ALL_TESTS();
}
```

代码中的`main()`函数首先初始化Google Mock，把命令行的参数传给它，然后运行所有的测试。

大多数时候，`main()`函数只需要这些。Google Mock也提供了一个默认的`main()`函数实现供你使用。参见http://code.google.com/p/googletest/wiki/Primer#Wrting_the_main()_Function获得更多信息。

1.6 CppUTest

除了Google Test/Google Mock，也可以选择CppUTest作为单元测试框架。CppUTest提供许多同样的特性，此外它还有一个内置的内存泄漏监测器。James Grenning的《测试驱动的嵌入式C语言开发》一书中提供了更多的示例。

1.6.1 安装 CppUTest

（注意：这里的步骤适用于CppUTest 3.3。包含大量改动的最新版本3.4，在本书截稿时才发布，所以本书并没有使用这一版。）

CppUTest的项目主页是http://www.cpputest.org/，可以从http://cpputest.github.io/cpputest/下载。下载好文件，解压至home目录下的新目录，如pputest。

创建环境变量CPPUTEST_HOME。如下：

```
export CPPUTEST_HOME=/home/jeff/cpputest
```

可以用make构建CppUTest。如果需要模拟支持，需要构建CppUTestExt。

```
cd $CPPUTEST_HOME
./configure
make
make -f Makefile_CppUTestExt
```

可以使用`make install`将CppUTest安装至/usr/local/lib下。

也可以通过CMake来构建CppUTest。

如果使用Windows，也可以找到一些使用MSBuild的批处理文件，适用于Visual Studio 2008和2010。

1.6.2　创建 main 函数以运行 CppUTest 测试

本书示例WAV Reader中的代码包含一个testmain.cpp文件，与CppUTest配合使用。

wav/1/testmain.cpp

```
#include “CppUTest/CommandLineTestRunner.h”

int main(int argc, char** argv) {
    return CommandLineTestRunner::RunAllTests(argc, argv);
}
```

1.7　libcurl

libcurl提供了客户端URL传输库，它支持HTTP和其他许多协议。它还支持cURL命令行传输工具。在本书中，用cURL指代这个库。

cURL的项目主页是http://curl.haxx.se/，可以从http://curl.haxx.se/download.html下载。下载文件后，解压至home目录下。创建环境变量CURL_HOME，如下：

```
export CURL_HOME=/home/jeff/curl-7.29.0
```

可以使用CMake构建这个库，如下：

```
cd $CURL_HOME
mkdir build
cd build
cmake ..
make
```

1.8　JsonCpp

JsonCpp提供了数据交换格式支持，如JavaScript Object Notation（JSON）。

JsonCpp的项目主页是http://jsoncpp.sourceforge.net/，可以从http://sourceforge.net/projects/jsoncpp/files/下载。下载文件后，解压至home目录下。创建环境变量JSONCPP_HOME，如下：

```
export JSONCPP_HOME=/home/jeff/jsoncpp-src-0.5.0
```

JsonCpp需要Scons，这是一个基于Python的构建系统。在Ubuntu下，使用下面的命令安装Scons:

```
sudo apt-get install scons
```

切换到$JSONCPP_HOME所指的目录，然后使用Scons构建库。

```
scons platform=linux-gcc
```

在OS X下，用linux-gcc作为平台选项在我的系统上工作。

在我的系统上，构建JsonCpp会生成$JSONCPP_HOME/libs/linux-gcc-4.7/libjson_linux-gcc-4.7_libmt.a。可以创建一个符号链接指向这个文件，如下：

```
cd $JSONCPP_HOME/libs/linux-gcc-4.7
ln -s libjson_linux-gcc-4.7_libmt.a libjson_linux-gcc-4.7.a
```

1.9 rlog

rlog为C++提供消息日志机制。

rlog的项目主页是http://code.google.com/p/rlog/。下载文件后，解压至home目录下。创建环境变量RLOG_HOME，如下：

```
export RLOG_HOME=/home/jeff/rlog-1.4
```

在Ubuntu下，可以使用下面的命令构建rlog库：

```
cd $RLOG_HOME
./configure
make
```

在OS X下，只有在打上一个补丁（patch）后才能编译rlog。参见https://code.google.come/p/rlog/issues/details?id=7获得更多的信息，以及补丁代码。我使用了第三条评论（This smaller diff...）中提及的代码。也可以在源代码发布包中找到这个补丁代码（code/wav/1/rlog.diff）。

可以使用下面的命令打补丁和构建rlog：

```
cd $RLOG_HOME
patch -p1 [path to file]/rlog.diff
autoreconf
./configure
cp /opt/local/bin/glibtool libtool
make
sudo make install
```

configure命令会复制一个名为libtool的二进制文件到rlog目录，但它不是rlog想要的。cp命令行所做的是复制glibtool覆盖libtool，来修复这个问题。

如果这个补丁不能工作，可以尝试手动修改。在$RLOG_HOME/rlog/common.h.in文件中，有这么一行：

```
#  define RLOG_SECTION  __attribute__((section("RLOG_DATA")))
```

用下面的代码替代这一行：

```
#ifdef _APPLE_
#  define RLOG_SECTION  __attribute__((section("__DATA, RLOG_DATA") ))
#else
#  define RLOG_SECTION  __attribute__((section("RLOG_DATA") ))
#endif
```

如果在构建rlog时还是遇到问题（这在Mac OS和Windows上的确非常困难），不要担心。在完成遗留代码示例时，跳至8.9节，学习怎样把rlog彻底从混乱中解救出来。

1.10　Boost

Boost提供了大量的C++基础库。

Boost的项目主页是http://www.boost.org，可以从http://sourceforge.net/projects/boost/files/boost下载。Boost会定期更新版本。下载文件，解压至home目录下。创建环境变量BOOST_HOME和BOOST_VERSION，如下：

```
export BOOST_ROOT=/home/jeff/boost_1_53_0
export BOOST_VERSION=1.53.0
```

许多Boost库只需要头文件。实现下面的步骤，可以构建所有使用Boost的示例，除了第8章中的示例。

为了构建第8章中的代码，需要构建和链接到Boost中的库。我使用下面的命令来构建相应的库：

```
cd $BOOST_ROOT
./bootstrap.sh  --with-libraries=filesystem,system
./b2
```

这里使用的命令应该可以工作，如果不能的话，请参见http://ubuntuforums.org/ showthread.php?t1180792（注意，`bootstrap.sh`的参数--with-library应该是--with-libraries）。

1.11　构建示例并运行测试

安装好所需的软件后，就能构建所有版本的示例，接下来就可以运行测试。在一个示例的版本目录下，首先需要使用CMake创建一个makefile：

```
mkdir build
cd build
cmake ..
```

遗留代码示例（参见第8章）使用Boost中的一些库，不仅仅是头文件。CMakeLists.txt使用BOOST_ROOT环境变量。原因有二：首先，通过`include_directories`指明，在哪里能找到Boost

头文件；其次，CMake通过执行`find_package`定位Boost库。

在构建遗留代码示例的时候，可能会遇到错误，表明找不到Boost。此时，可以尝试在执行CMake时传入BOOST_BOOT改变位置，如下：

```
cmake -DBOOST_ROOT=/home/jeff/boost_1_53_0 ..
```

否则，确保Boost库已经被正确构建。

一旦使用CMake创建完makefile后，就可以切换到示例的build目录，执行下面的命令构建示例：

```
make
```

为了执行测试，切换到示例的build目录，使用下面的命令执行：

```
./test
```

对于第7章中的库示例，可以在build/library/Tests目录下找到测试执行文件。

1.12 结束语

在本章中，你学习了构建和运行本书示例需要做的事情。记住，最好的学习方式就是跟随示例，动手去做。

如果在配置的时候遇到了困难，首先向可信的同伴寻求帮助。多一双眼睛可以快速发现已困扰你一段时间的问题。可以访问本书的主页http://pragprog.com/titles/lotdd，找到有帮助的提示，或者去论坛讨论。如果你和同伴都被难住了，请给我发电子邮件吧！

第 2 章

测试驱动开发：第一个示例

2.1　开场白

写个测试，保证它通过，接着重构设计。这就是TDD的全部内容。但是这三个简单步骤的背后却另有乾坤。理解怎样利用TDD将使你受益匪浅。俗话说“前车之鉴，后事之师”，如果缺乏前人的经验，你也很有可能放弃TDD。

与其让你在学习中趔趄前行，我更愿意带领你以测试驱动的方法开发一些代码，这将帮助你理解每一步背后的故事。在本章中，最好的学习方法就是跟着示例一起做。当然首先要确保开发环境已设置好（参见第1章）。

虽然书中示例规模不大，但也绝非一点用没有或毫无价值（当然也不是什么高科技）。这些示例提供了许多教学点，展示了TDD怎样帮助你增量地设计一个算法。

好了，希望你已经准备好编写代码了！

2.2　Soundex 类

在许多应用程序中，搜索是很常见的功能。一次有效的搜索应该找出匹配的结果，即便用户的输入有拼写错误。许多人就曾以各种奇葩的方式拼错了我的名字：Langer、Lang、Langur、Lange、Lutefish就是其中一些。不管怎样，还是希望他们能够找到我。

本章将以TDD方法开发Soundex类，该类能提升应用程序的搜索能力。这个算法是将单词编码为一个字母和三个数字，它将发音相似的单词映射到相同的编码。下面是维基百科上关于Soundex规则的描述。[1]

(1) 保留第一个字母。丢掉所有出现的a、e、i、o、 u、y、h、w。

(2) 以数字来代替辅音（第一个字母除外）：

---

① http://en.wikipedia.org/wiki/Soundex

- b、f、p、v : 1
- c、g、j、k、q、s、x、z : 2
- d、t : 3
- l : 4
- m、n : 5
- r : 6

(3) 如果相邻字母编码相同，用一个数字表示它们即可。同样，如果出现两个编码相同的字母，且它们被h或w隔开，也这样处理；但如果被元音隔开，就要编码两次。这条规则同样适用于第一个字母。

(4) 当得到一个字母和三个数字时，停止处理。如果需要，补零以对齐。

2.3 开始吧

关于TDD的一个常见误解是，在实现产品代码前先定义好所有的测试。事实正好相反，每次你只需要关注一个测试，此后逐渐考虑下一个加入到系统中的行为。

TDD的一般方法是逐步实现下一个最简单的规则（想要了解特定的形式化方法，请参考10.4节）。哪个有用行为的实现最直接且需要最少的代码改动？

基于这一思路，要从哪里开始实现Soundex呢？让我们快速推想一下实现每个规则都需要做些什么。

规则3似乎是最常用到的。规则4指示何时停止编码，这条规则应该只在其他规则生成编码时才会用到。规则2暗示第一个字母已经处理完毕，所以从规则1着手。它似乎比较直接。

规则1告诉我们，保留名字的第一个字母，然后……停止！让我们尽可能地保持事情简单。如果一个单词中只有一个字母呢？为此写个测试。

c2/1/SoundexTest.cpp

```
Line 1 #include "gmock/gmock.h"
     2 TEST(SoundexEncoding, RetainSoleLetterOfOneLetterWord)  {
     3    Soundex soundex;
     4 }
```

测试列表

在TDD过程中，你编写并通过的每一个测试，都代表新加入到系统中的一种行为。除了完成整个功能外，通过的测试数目是衡量进度的最佳指标。每个测试都代表系统中一个小的行为。

虽然不能事先知晓所有需要编写的测试，但需要对将要处理的事务有一些初步的认识。许多使用TDD方法的开发者将他们想到的测试记录到**测试列表**中（测试列表在*Test Driven*

Development: By Example [Bec02]中首次提出）。这个列表包含测试的名称，或在需要做代码清理时提示。

可以把测试列表写在工作台边上的写字板上（也可以把它作为注释写入测试文件，只要在提交代码前删掉就行）。这个列表仅属于你自己，所以如果喜欢的话，大可以使用简要或隐晦的表达方式。

在做测试驱动开发并思考新的测试用例时，记得把它们加到测试列表里。当添加一些你认为将来需要清理的代码时，也在这个列表中加一个提醒项。在完成一个测试或任务时，把它从列表中删除即可，就这么简单。如果在编码结束后，发现仍有没完成的任务项，可以把它们加入到下一个编码阶段的列表。

可以把测试列表作为初始设计的一部分。它能帮助说明你认为自己需要构建什么。它也可以启发你去思考其他需要做的事情。

不要被这个列表束缚，它决定不了你要做什么，也决定不了你做事的顺序。但是，TDD是一个自然的流程，通常要顺着测试指引的方向去做下一件事。

在学习TDD时，管理测试列表非常管用。试一试吧！

- 第1行代码中包含了gmock头文件，它具备写一个测试所需的全部功能。
- 一个简单的测试声明需要使用TEST宏（第2行代码所示）。TEST宏有两个参数：测试用例的名称和测试的描述性名称。根据Google的文档，测试用例（test case）是一些能共享数据和子程序的测试集合。[①]（这个术语在这里被复用了，对于有些人而言，一个测试用例代表一种情境。[②]）
  从左到右阅读测试用例名称和测试名称，可以连成一句话，描述了我们想要验证的行为：Soundex encoding retains [the] sole letter of [a] one-letter word。因为还要为Soundex编码行为写其他测试，所以用SoundexEncoding作为测试用例名字，以帮助组织相关测试。
  不要低估好的测试名称的重要性。如下所示。

测试名称的重要性

多留心一下命名。长久看来，花一点精力起一个描述性强的测试名称是值得的，这是因为维护代码的人需要经常阅读测试。好的测试名称同样会帮助你自己（写测试的人）更好地理解将要构建的东西背后的意图。

① http://code.google.com/p/googletest/wiki/V1_6_Primer#Introduction:_Why_Google_C++_Testing_Framework?
② Test Case原意是指一个测试条件集合，这个条件集合定义了系统在此集合下的行为。但在Google的文档中，术语的意思被重载为测试集合。作者说“对于一些人来说，它表示一种测试情境”，指的正是此术语的原意。——译者注

> 你将为系统中的每个新行为开发一些测试。将测试名称视为一种索引，它可以为开发人员快速提供一个有关代码行为的准确描述。测试名称越容易理解，你和其他开发人员就能越快找到需要的东西。

- 第三行代码中，我们创建一个Soundex对象，然后……停止！在写更多的测试之前，我们知道已经加入了一些不能通过编译的代码：我们还没有定义Soundex类。在继续编写测试前，先停下来去解决这个问题。这个方法和Bob大叔关于TDD的三条规则保持一致。
  - 只在为了使失败测试通过时才编写产品代码。
  - 当测试刚好失败时，停止继续编写。编译失败也是失败。
  - 只编写刚好能让一个失败测试通过的产品代码。

  （Bob大叔就是Robert C. Martin。可以参考3.4节了解更多信息。）

  在使用C++时，增量地获得反馈是很好的方法，因为有时候只需几行测试代码，就能产生一大堆编译错误。若能及时看到所写代码产生的错误，那么将会更容易地解决它们。

  撇开TDD的三条原则不谈，有时候你会觉得，在运行测试之前编写完整测试更靠谱，或许这有助于更好地理解应该怎样设计待测试的接口。你或许也会觉得不值得花费额外的编译时间，来获得更及时的反馈信息。

  但是现在，尤其在学习TDD时，及时获得反馈很有用。归根结底，还是由你决定怎样增量地设计每个测试。

编译器显示我们需要Soundex类。可以为Soundex添加一个编译单元（一对.h/.cpp文件），但是先别自找麻烦。相反，不要急于使用独立文件，先简单地在包含测试的文件中声明[①]所有的东西。

在准备提交代码，或苦于所有东西都放在一个文件中的时候，再以适当的方式把测试从产品代码中剥离。

c2/2/SoundexTest.cpp

```
➤ class Soundex{
➤ };

  #include "gmock/gmock.h"

  TEST(SoundexEncoding, RetainSoleLetterOfOneLetterWord)  {
     Soundex soundex;
  }
```

① 作者在书中许多地方未区分声明（declare）和定义（define）。不过这只是一个小细节，并不影响主题的论述。读者在阅读时可以依据上下文作出判断。——译者注

提问：把所有东西放在一个文件中不是危险的捷径吗？

回答：这是一种短期内不引入复杂开销而节省时间的方式。前提是稍晚些分开文件产生的开销，小于一直在文件间来回切换的开销。当以TDD的方式为一个新行为打磨设计时，很可能经常改变接口。过早地拆分出头文件往往只会成为累赘。

说到“危险”，你忘记过在提交代码前拆分文件吗？

提问：但是遵循TDD周期的话，你不是应该要清理代码的吗？你不是一直要保证代码尽量维持在最高质量吗？

回答：一般而言，这两个问题的答案是肯定的。但是我们的代码没问题，我们只是在确实需要的时候，才选择一个更有效的组织方式。在真的需要之前，我们推延了做这种复杂的事情，否则过早地引入这些复杂性往往会使开发速度变慢。（一些敏捷支持者使用缩写YANGI——You ain’t gonna need it。）

如果这种理念深深地困扰你，就马上把东西分散到不同的文件吧！你依然能够进行其余的练习。但是，我建议先试一试这种方法。TDD以一种安全的方式磨砺你，所以要敢于尝试你认为可能更有效的工作方式。

我们遵循了第三条规则：只编写刚好让一个失败测试通过的产品代码。很显然，我们还没有完成测试。RetainsSoleLetterOfOneLetterWord并没有测试任何行为，所以它验证不了什么。然而，我们却可以对出现的任何负反馈[1]及时采取应对措施（这里的情况是编译失败），加入足够的代码消掉它。为使编译通过，我们加入了Soundex类的一个空定义。

这个时候构建并执行测试就能得到正反馈了。如下：

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SoundexEncoding
[ RUN      ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord
[       OK ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord (0 ms)
[----------] 1 test from SoundexEncoding (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

欢乐时刻！

等等，还不算。毕竟，这个测试除了构造一个Soundex类的实例（Soundex还是个空类），没做任何事情。但是，我们已经有了一些基本的要素。更重要的是，我们已经证明了目前所做的是正确的。

你进行到这一步了吗？你有没有拼错include的文件名，或者忘记在`class`定义后加上分号？如果有，恭喜你！你已经以最少的代码制造出了错误。在TDD中，如果践行这种安全编码方式的

① 负反馈即为任何错误。与之对应的是正反馈。——译者注

话，即尽早并频繁地测试，通常测试失败的原因只有一个。

测试通过了，是时候将代码提交到本地了。你有合适的工具吗？一个好的版本控制系统，让你可以在代码状态好（或者说，所有测试都通过）的时候方便地提交代码。如果稍后遇到麻烦的话，很容易将代码回滚到一个已知的完好状态，重新尝试改动。

TDD的一部分理念是，每个通过的测试都代表了加入到系统中的、已经验证了的行为。当然，这并不一定意味着可以发布代码了。但是，更多地以此增量的方式思考和频繁地集成本地已经验证通过的功能，成功的几率就会大大增加。

继续吧！往测试中加入一行代码，来表明我们期待的客户端代码与Soundex对象的交互方式。

c2/3/SoundexTest.cpp

```
TEST(SoundexEncoding, RetainSoleLetterOfOneLetterWord)  {
   Soundex soundex;

➤  auto encoded = soundex.encode("A");
}
```

我们在添加测试的同时也在作决定。如上述代码所示，我们决定Soundex类暴露一个名为`encode()`的公共成员函数，它有一个字符串类型的参数。尝试编译会导致失败，因为`encode()`尚不存在。这一负反馈迫使我们去写足够的代码，让测试通过编译，然后运行。

c2/4/SoundexTest.cpp

```
class Soundex
{
➤ public:
➤    std::string encode(const std::string& word) const {
➤       return "" ;
➤    }
};
```

现在代码可以编译了，所有的测试也可以通过，这还算不上非常有趣的时刻。是时候验证一些有用的东西了：给定一个字母A，`encode()`能返回正确的Soundex代码吗？我们用一个断言（assertion）来表达这一关注。

c2/5/SoundexTest.cpp

```
TEST(SoundexEncoding, RetainSoleLetterOfOneLetterWord)  {
   Soundex soundex;

   auto encoded = soundex.encode("A");

➤  ASSERT_THAT(encoded, testing::Eq("A"));
}
```

断言可用来验证结果是否符合预期。上面代码中的断言声明了encode()返回的字符串等于指定的字符串。现在，编译是通过了，但是断言却失败了。

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SoundexEncoding
[ RUN      ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord
SoundexTest.cpp:21: Failure
Value of: encoded
Expected: is equal to 0x806defb pointing to "A"
  Actual: "" (of type std::string)
[  FAILED  ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord (0 ms)
[----------] 1 test from SoundexEncoding (0 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord 1 FAILED TEST

 1 FALED TEST
```

乍一看，Google Mock给出的相关输出信息可能不太容易读懂。从最后一行开始看吧！如果是PASSED，可以不用看测试的输出，因为所有测试都过了！如果是FAILED（正如示例一样），我们可以看到有多少测试失败了。如果是其他类别的信息，则是测试程序在一个测试运行过程中崩溃了。

如果有一个或多个测试失败了，可以从下至上观察Google Mock输出信息，找到失败的测试。Google Mock会在每个测试名称前输出一个`[RUN]`，在测试失败的情况下输出`[FAILED]`或`[OK]`。失败的时候，`[RUN]`和`[FAILED]`之间的信息行可以帮助我们了解测试为什么失败。在上面这个示例中，可以看到下面信息：

```
[ RUN      ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord
SoundexTest.cpp:21: Failure
Value of: encoded
Expected: is equal to 0x806defb pointing to "A"
  Actual: "" (of type std::string)
[  FAILED  ] SoundexEncoding.RetainsSoleLetterOfOneLetterWord (0 ms)
```

这个断言失败的意思是，Google Mock期待一个名为encoded的局部变量的值为`"A"`，但实际的值是一个空字符串。

这个断言失败是意料之中的，因为为了通过编译，我们特意硬编码了一个空字符串。得到这个负反馈其实是好事，而且这也属于TDD周期内可以发生的事情。首先，我们正是想确保新加的断言不能通过，它代表了还没实现的功能。（有时候是通过的，通常这不是个好事情，参考3.5节）。我们也想确保测试是有效的。起初测试失败，在加入适当的代码后通过了，这也说明测试是可靠的。

失败的测试提醒我们编写仅够通过断言的代码即可。如下：

c2/4/SoundexTest.cpp

```
std::string encode(const std::string& word) const {
➤   return "A";
}
```

现在编译并重新运行测试。最后两行的输出显示测试通过。

```
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

发布吧！

我是在开玩笑吗？好吧，不是的。我们想以渐进的方式工作。这么说吧：如果某人告诉我们去构建一个Soundex类，而且只支持字母A即可，那我们已经完成了。自此，我们还想把代码弄得更干净些，但仅此而已，不需要额外的处理逻辑。

另外一个说法是，测试表明了目前系统拥有的功能。现在只有一个测试。为什么还要添加超出此测试所测行为之外的代码呢?

当然，还没完成。我们有大量的其他需求，这些需求将用TDD方法增量地加入系统。就目前已有的测试来说，我们也没完成。我们必须清理掉自己制造的一些小凌乱。

2

2.4 去掉不干净的代码

什么？我们只写了一行产品代码和三行测试代码就有问题了啊？当然，即便区区几行代码也非常容易引入缺陷。TDD方法提供了这样的契机，当一些小问题出现时，我们可以及时修复，这样就避免小问题越积越多（甚至产生一些大问题）。

我们先审阅一下刚写的测试和产品代码，找一找缺陷吧！有一点是可以确定的，测试中的断言不是非常便于阅读。

```
ASSERT_THAT(encoded, testing::Eq("A"));
```

和测试声明（测试用例和测试名称的组合）类似，我们也希望断言读起来像个句子。为此，我们引入using指示符来帮忙。

c2/7/SoundexTest.cpp

```
#include "gmock/gmock.h"
➤ using ::testing::Eq;

TEST(SoundexEncoding, RetainSoleLetterOfOneLetterWord){
   Soundex soundex;
   auto encoded = soundex.encode("A");
➤  ASSERT_THAT(encoded, Eq("A"));
}
```

现在可以通顺地朗读我们的断言了：断言encoded的值等于"A"。

我们称这一小改动为重构，重构是一种代码改写，其特点是在保持现有行为不变的前提下改进设计。这里，我们通过增强测试的表达力来提升测试的设计。Eq()所在的命名空间是一个与测试本意无关的实现细节。隐藏这一细节可以提升测试中的抽象程度。

另一个难点是，我们常常要面对重复代码。一旦有大量的重复代码，维护成本和风险就会大大增加。

我们的Soundex类还没有明显的重复代码。但是，一并看一下测试代码和产品代码，就会发现一个公共的常量，字符串"A"。我们想去除这一重复代码。还有一个问题，测试名称（RetainsSoleLetterOfOneLetterWord）声明了一个一般的行为，但是代码实现仅仅支持一个特定的单个字母。我们想找一个一箭双雕的方法去掉这个硬编码的"A"。

直接返回传入的字符串可以吗？

c2/8/SoundexTest.cpp

```
class Soundex
{
public:
   std::string encode(const std::string& word) const {
➤     return word;
   }
};
```

任何时候，一个完整的测试集合声明了系统中期望的行为。这里蕴含着一个潜台词：如果一个行为没有对应的测试来描述，那这个行为要么不存在，要么不是期望的（或者测试本身没有尽到描述行为的职责）。

那该怎么办呢？现在有一个测试。但它只支持单个字母的单词。所以，可以假定Soundex类仅需要支持单个字母的单词，至少目前是这样。如果所有的单词都只有一个字母，那么最简单的通用方案就是直接返回传给encode()的单词。

[其他TDD流派会想，如果不这样做，编出来的代码会是什么样的？一个可行的技巧是Triangulation[①]（参见10.4.2节），写一个类似的断言，但是有不同的数据期望。你会在书中发现更多可行的方法，但是目前先使事情保持简单。]

刚才的改动都很小且琐碎，但是当下完成这些改动是合时宜的。TDD中的重构步骤给予我们关注所有问题的契机，不论问题是大还是小，都是由小的、独立的代码改变引起的。当历经TDD的各个周期时，我们会使用重构来审阅设计，同时修复出现的所有问题。

重构的主要关注点是提升表达能力，去除重复代码。就代码的可维护性来说，这两个点最有裨益。在本书的后面章节，我们将使用一些其他的设计知识，譬如SOLID设计原则和代码坏味。

① *Test Driven Development: By Example* [Bec02]

2.5 增量性

2

问答时间到了！

提问：你真的对明知道将要替换的东西进行硬编码吗？

回答：我总是被问到这个问题。答案是肯定的。

提问：这似乎不明智！

回答：这不是个问题，但是在第一次接触这种方式时，认为它不明智是正常的。我也曾经有过这样的感受。我是过来人。

提问：你会一直以此方式工作吗？如果对所有东西硬编码，那怎么能够搞定所有事情？

回答：这是两个问题，但我很乐意一并回答！是的，我们会一直以增量的方式工作。这个技巧能快速让测试通过。不要担心，硬编码的东西最多存在一小会。我们心里清楚离目标还有一段距离，所以需要写更多的测试来描述这些行为。在这个例子中，还要实现其他的规则。在接下来编写其他测试时，我们将会用有趣的逻辑替换掉硬编码，使测试通过。

增量性是TDD取得成功的关键。乍一看，增量方法显得非常不自然且速度慢。但是，随着时间的推移，小步的增量开发反而能够加快你的速度，部分原因是这可以避开由一次编写而成的大量且复杂的代码产生的错误。坚持它！[1]

精明的读者或许注意到了，我们编写的代码并不完全符合Soundex的规范。规则4的最后说，如果没有三个数字，需要补零。喔，这就是规范的乐趣所在！我们必须认真且全面地阅读它，彻底理解各个部分是怎样交互的。（最好与用户沟通，他会阐明哪些是期望的。）目前来说，规则4和我们所实现的代码尚不吻合。

想象一下，这些规则是逐个向我们提出的。“先把规则1的第一部分完成，然后再给你下一条规则。”TDD与后一种方法一致，规范的各个部分是增量地编写进系统。这一方法可以使我们以任何顺序一点点地开发系统，并持续地验证，向前推进。这里有一个权衡：相比于多做一点计划而言，我们可能要花费更多的时间合并新的代码。稍后再来讨论这个问题。目前来说，先看看没有这个权衡，会发生什么。

现在有两个工作要做：为新行为写一个新测试；修改已有的测试以满足规范。下面是新测试：

c2/9/SoundexTest.cpp

```
TEST(SoundexEncoding, PadsWithZerosToEnsureThreeDigits)  {
   Soundex soundex;
```

① 小步开发出来的代码更容易适应未来的需求和变化。——译者注

```
   auto encoded = soundex.encode("I");

   ASSERT_THAT(encoded, Eq("I000"));
}
```

（一个审阅人提出问题："为什么在编写测试前不认真地阅读Soundex规则呢？"好问题！的确，我们没有认真阅读规范。TDD的一个强项就是在信息不完全的情况下，我们依然可以向前推进，并且可以在得到新信息后及早纠正之前的代码。）

加入的每个测试都是独立的。我们不会将一个测试的结果作为另一个测试的前提。每个测试必须设置好自己的上下文。所以，新加的测试需要创建自己的Soundex类实例。

运行测试后，测试结果提示，`encode()`返回的是`"I"`而非`"I000"`。让它通过很简单。

c2/9/SoundexTest.cpp

```
std::string encode(const std::string& word) const {
➤  return word + "000";
}
```

硬编码可能再次导致事情乱作一团，但它能帮助我们不跑偏。就我们目前的测试而言，Soundex类不需要额外的行为。此外，通过采取尽可能小的步伐，我们在往系统中加入新行为时，必须编写额外的测试。

新的测试通过了，但是第一个测试失败了。这是因为测试描述的行为和维基百科上列出的规范不吻合。

如果一个测试通过了，就说明这个测试正确地描述了系统是如何工作的。如果测试设计得好，那么会起到例子的作用，比规范更易读。在接下来的练习中，我们会继续专注让测试具有可读性（我偶尔甚至把它们称之为规范）。

c2/9/SoundexTest.cpp

```
TEST(SoundexEncoding,RetainsSoleLetterOfOneLetterWord){
   Soundex soundex;

   auto encoded = soundex.encode("A");

➤  ASSERT_THAT(encoded, Eq("A000"));
}
```

如上述代码所示，让第一个测试通过并不难！

现在有两个大体相同的测试，只是数据稍微有点差别。这没关系。每个测试分别描述一种行为。我们不仅要确保系统按预期工作，还要让每个人知道所有既定的系统行为。

是时候重构了。`encode()`中的代码在描述其中可能发生的操作时有点模糊其词。我们决定提取独立的方法（Method），配以意图明确的名字。

c2/10/SoundexTest.cpp

```
public:
   std::string encode(const std::string& word) const {
➤     return zeroPad(word);
   }

   private:
➤     std::string zeroPad(const std::string& word) const {
➤        return word + "000";
➤     }
```

2.6 fixture 与设置

在重构的时候，不仅要审阅产品代码，还要审阅测试。如上所述，我们的测试都需要创建Soundex类实例，并且使用相同的代码。我们不乐意看到此类貌似无关紧要的重复代码。这些重复会积累得很快，并且通常会演变为更复杂的重复代码。这也会让测试变得有点主次不分，对于阅读代码的人来说，这会分散注意力，从而忽视真正需要关注的重要内容。

相关测试拥有一些共同的代码是常见的。Google Mock允许我们定义一个fixture类，我们可以在这个类中为相关的测试声明函数和数据。(从技术角度说，所有的Google Mock测试都使用一个由Google Mock自己生成的fixture。)

c2/10/SoundexTest.cpp

```
➤ class SoundexEncoding : public testing::Test  {
➤ public:
➤    Soundex soundex;
➤ };

➤ TEST_F(SoundexEncoding, RetainsSoleLetterOfOneLetterWord)  {
     auto encoded = soundex.encode("A");

     ASSERT_THAT(encoded, Eq("A000"));
  }

➤ TEST_F(SoundexEncoding, PadsWithZerosToEnsureThreeDigits)  {
     auto encoded = soundex.encode("I");

     ASSERT_THAT(encoded, Eq("I000"));
  }
```

我们在上述代码中创建了一个名为SoundexEncoding的fixture(必须从::testing::Test继承)。这样，创建Soundex类实例就在一个地方完成了。在fixture内部，我们声明了公共成员变量`soundex`，以便测试可以访问。(如果你对暴露变量`soundex`有点担心的话，记住fixture类只在这个.cpp文件中。我们想避免在测试中加入多余的杂乱代码。)

Google Mock会在运行每个测试时创建fixture类实例。在Google Mock运行RetainSole-

LetterOfOneLetterWord前，它先创建一个SoundexEncoding类实例。在运行PadsWithZeroToEnsureThreeDigits前，创建另外一个SoundexEncoding类实例。为了使用自定义的fixture，我们把TEST宏替换为TEST_F，F表示fixture。如果忘记使用TEST_F，任何使用了fixture类成员的测试代码都会编译失败。

现在，可以删掉测试内的局部变量soundex，因为每个测试都可以访问fixture中的成员。我们增量地作出这些代码改动。在定义完fixture类并修改宏处理后，我们从第一个测试中删掉局部变量soundex。运行所有的测试来验证这个改动，继而删掉第二个测试中的局部变量soundex，然后再次运行所有的测试。

去掉测试中重复的Soundex类实例定义至少会产生下面两个影响。

- 提升了测试的抽象度。现在，每个测试中只包含两行代码，这有助于我们集中精力关注与测试相关的东西。我们也看不到Soundex类实例是怎样构造出这一不相干的细节的（参见7.4节，详细了解为什么这很重要）。
- 可以降低未来维护测试的开销。试想一下我们必须改变Soundex类实例的构造方式（譬如我们需要能够将语言种类作为一个构造函数的参数）。将Soundex类实例的构造放到fixture中，这意味着改动一个地方即可。否则，要改动每个测试。

每个测试只有两行代码，这让测试更具可读性。还可以做些什么呢？我们可以让每个测试只有一行代码，同时保持可读性。另外，我也不是显式使用using指示符的拥趸，所以也可以去掉它。

c2/11/SoundexTest.cpp

```
  #include "gmock/gmock.h"
➤ #include "Soundex.h"

➤ using namespace testing;

➤ class SoundexEncoding: public Test {
  public:
     Soundex soundex;
  };

  TEST_F(SoundexEncoding, RetainsSoleLetterOfOneLetterWord) {
➤    ASSERT_THAT(soundex.encode("A"), Eq("A000"));
  }

  TEST_F(SoundexEncoding, PadsWithZerosToEnsureThreeDigits) {
➤    ASSERT_THAT(soundex.encode("I"), Eq("I000"));
  }
```

从上述代码中可以看到，测试仅仅引用了Soundex.h头文件。短期内，测试和产品代码同在一个文件是有益的。但现在，我们需要在一个文件中不断地拉上拉下，所以有点费力。先把测试和头文件分开（稍后再决定是否应该创建.impl文件），下面是头文件。

c2/11/Soundex.h

```
➤ #ifndef Soundex_h
➤ #define Soundex_h
  #include <string>

  class Soundex
  {
  public:
     std::string encode(const std::string& word) const {
        return zeroPad(word);
     }

  private:
     std::string zeroPad(const std::string& word) const {
        return word + "000";
     }
  };

➤ #endif
```

2.7 思索与测试驱动开发

简单地说，TDD的周期就是写一个测试，先确保测试失败，然后编码让测试通过，接着审阅代码和打磨设计（包括测试的设计），最后确保所有测试依然通过。在一天的工作中，你不断地重复此周期，保持周期短小，以便得到最多的反馈。虽然是重复，但绝非盲目，每一个周期你都需要思考很多事情。3.3节列出了每一步需要回答的问题。

为了保持事情持续推进，假定你遵从了上面提到的步骤，我只会偶尔提醒一下。你可以在显示器边上贴一个便签用作提醒。

在下一个测试中，我们将处理规则2（即在第一个字母后，用数字替换辅音）。替换规则表中说字母b对应数字1。那就先写个这样的测试，如下：

c2/12/SoundexTest.cpp

```
  TEST_F(SoundexEncoding, RetainsSoleLetterOfOneLetterWord)  {
➤    ASSERT_THAT(soundex.encode("Ab"), Eq("A100"));
  }
```

不出意料，测试失败了。

```
Value of: soundex.encode("Ab")
Expected: is equal to 0x80b8a5f pointing to "A100"
  Actual: "Ab000" (of type std::string)
```

如大部分要写的测试一样，对应的解决方案可能枚不胜举，但可能只有少量是合理的。从技术角度讲，唯一要做的是让这个测试通过，然后打磨我们的解决方案。

但是，我们寻求的是通用的解决方案（也不要过于通用，以至于考虑过多），也不要对已处

理过的行为重复编码。

可以提供一个让测试通过的解决方案，如下：

```
std::string encode(const std::string& word) const {
   if (word == "Ab") return "A100";
   return zeroPad(word);
}
```

然而，对于用数字替换辅音而言，这一段代码不是通用的解决方案。它也引入了一些重复的东西：特例"Ab"的结果为"A1"的补零对齐输出，即"A100"，但是我们已经有了对任何词做补零操作的通用代码。

你可能认为这个理由不太充分，或许是吧。如果不限定时间，你能给出的替代方案数目可能也是无穷无尽的。但是TDD不是纯粹的科学，相反，可以把它想成软件工匠用来增量开发代码的工具。这个工具适合持续的实验、发现和改进。

行动胜于空谈！下面是迈向通用解决方案的一步：

c2/13/Soundex.h

```
std::string encode(const std::string& word) const {
➤    auto encoded = word.substr(0, 1);
➤
➤    if (word.length() > 1)
➤       encoded += "1";
➤    return zeroPad(encoded);
}
```

运行一下，新的测试没有通过。

```
Expected: is equal to 0x80b8ac4 pointing to "A100"
  Actual: "A1000" (of type std::string)
```

补齐逻辑不对。必须修改它，以考虑到待编码词的长度。

c2/14/Soundex.h

```
std::string zeroPad(const std::string& word) const {
➤    auto zerosNeeded = 4 - word.length();
➤    return word + std::string(zerosNeeded, '0');
}
```

测试通过了。太棒了！但是代码看起来变得有点乏味。当然，我们知道自己编写的encode()是怎样工作的。但是其他人可能需要花更多时间，仔细阅读代码，以理解其背后的意图。其实我们可以做得更好。先把它重构为更具可读性的方案吧，如下：

c2/15/Soundex.h

```
class Soundex
{
public:
```

```
   std::string encode(const std::string& word) const {
➤     return zeroPad(head(word) + encodedDigits(word));
   }

private:
➤  std::string head(const std::string& word) const {
➤     return word.substr(0, 1);
➤  }

➤  std::string encodedDigits(const std::string& word) const {
➤     if (word.length() > 1) return "1";
➤     return "";
➤  }

   std::string zeroPad(const std::string& word) const {
   // ...
   }
};
```

瞧，我们正一点点地完善Soundex编码算法。同时，重构有助于确保核心算法清晰，实现细节也很整洁。

以声明性的方式组织代码，使其非常易于理解。设计中非常重要的一方面是从实现（怎么做）中分离接口（做什么），这提供了迈向更高层次设计方案的跳板。每到TDD的重构环节时，都要考虑类似的重组。

有些读者或许担心实现细节。第一，是不是应该用stringstream，而不是直接将字符串连接起来？第二，为什么不尽可能地用单独的`char`？例如，为什么用`return words.substr(0, 1);`而非`return word.front();`？第三，用`return std::string();`不是比`return "";`更好吗？

这些替代的代码方案可能更好。但这些都是过早优化（premature optimization）。这个时候，一个好的设计（接口一致且代码可读性高）更重要。一旦以牢靠的设计实现了正确的行为后，再考虑是否优化性能（若没有事先做测量，先不要考虑优化，参考10.2节）。

撇开过早的性能优化不谈，我们的代码也确实需要雕琢一番。先去掉使用神奇的常量表示Soundex编码最大长度而产生的代码坏味，取而代之，使用一个具有合理名字的常量。

c2/16/Soundex.h

```
static const size_t MaxCodeLength{4};
// ...
std::string zeroPad(const std::string& word) const {
➤  auto zerosNeeded = MaxCodeLength - word.length();
   return word + std::string(zerosNeeded, '0');
```

你怎么看待`encodeDigits()`中硬编码的字符串`"1"`呢？我们的代码需要将字母b替换为1，所以不能用变量去掉这个硬编码。我们可以引入另一个常量，或从一个名副其实的函数中返回一个常量。

甚至可以先不管这个硬编码，待到下一个测试时再消除它。但是我们能保证在集成代码前写下一个测试吗？万一我们转去做其他事情了呢？那个时候再回头看这段代码，或许要花费更多的时间来理解它了。请保持增量交付的态度，现在就来解决问题！

c2/17/Soundex.h

```
  std::string encodedDigits(const std::string& word) const {
➤    if (word.length() > 1) return encodedDigit();
     return "";
  }

➤ std::string encodedDigit() const {
➤    return "1";
➤ }
```

2.8　测试驱动与测试

我们要以测试驱动方法开发更多的辅音变换逻辑（如c对应2、d对应3，等等），使解决方案更具通用性。为此，应该在ReplacesConsonantsWithAppropriateDigits中加一个断言，还是再创建一个测试呢？

TDD的经验法则是一个测试一个断言（参考7.3节，获取更多信息）。我们提倡专注测试行为，而非测试功能函数。大部分时候要遵从这一个规则。

对第二个辅音进行编码的断言不像是另一个行为。如果创建一个这样的测试，该怎么命名呢？难道是ReplacesBWith1、ReplacesCWith2……？

向测试中加入第二个断言来表示一个测试用例的情况比较少。我们希望一个断言失败时，其他的代码能继续执行。为此，我们使用Google Mock提供的`EXPECT_THAT`宏而非`ASSERT_THAT`。

c2/18/SoundexTest.cpp

```
  TEST_F(SoundexEncoding, ReplacesConsonantsWithAppropriateDigits) {
➤    EXPECT_THAT(soundex.encode("Ab"), Eq("A100"));
➤    EXPECT_THAT(soundex.encode("Ac"), Eq("A200"));
  }
```

上述代码中，第二个辅音处理情形需要一个`if`分支来处理特殊的情况。

c2/18/Soundex.h

```
  std::string encodedDigits(const std::string& word) const {
➤    if (word.length() > 1) return encodedDigit(word[1]);
     return "";
  }
➤ std::string encodedDigit(char letter) const {
➤    if (letter == 'c') return "2";
     return "1";
  }
```

再加上第三种数据输入情形。

c2/19/SoundexTest.cpp

```
TEST_F(SoundexEncoding, ReplacesConsonantsWithAppropriateDigits) {
   EXPECT_THAT(soundex.encode("Ab"), Eq("A100"));
   EXPECT_THAT(soundex.encode("Ac"), Eq("A200"));
➤  EXPECT_THAT(soundex.encode("Ad"), Eq("A300"));
}
```

第三个辅音处理情形让事情变得明朗，需要用一个基于哈希（hash）的集合代替简单的`if`分支。

c2/19/Soundex.h

```
std::string encodedDigit(char letter) const {
➤  const std::unordered_map<char,std::string> encodings {
➤     {'b', "1"},
➤     {'c', "2"},
➤     {'d', "3"}
➤  };
➤  return encodings.find(letter)->second;
}
```

现在，需要写代码支持其余的辅音转换。问题是，每个辅音转换功能都需要以测试驱动方式开发吗？

在TDD诞生初期有个口头禅："可能出问题的地方都需要测试。"人们常常会问："我需要测试什么？"这个口头禅算是一个漫不经心的回复。事实上，`encoding`中的映射（map）操作出现问题的风险比较小。不挨个测试一遍，应该也不会出什么问题。

一个相反的观点是，什么都可能出问题，无论它多么简单（我以前就是这样的，嘿嘿）。对重复数据越是感到乏味，犯错误的几率越大，甚至不会注意到已经犯错了。测试可以降低犯错的几率。

另外，测试好比一个辅音转换为数字的清晰文档（虽然你有可能辩称表格本身就是一个再清楚不过的文档了）。反过来，如果创建一个有几百个元素的表格，为每个都提供测试似乎很荒唐。

正确答案是什么？考量的重点在于我们在做*测试驱动开发*，而非*测试*。你或许会问：这有什么不同吗？答案是肯定的。用测试的技巧，你会全面地分析规范（也有可能是代码），并创建大量的测试来罗列各种行为。TDD则着力于代码设计。测试主要用于表述你要构建的行为。TDD过程中编写的测试大都是这个流程的附属产物。有了这些测试，在接下来改动代码时，你会更有信心。

其实，TDD与测试之间的区别很微妙。TDD的一个重要方面就是秉承够用心态。你写的测试只是为开发代码做准备。当要开发下一个行为时，再写测试。如果代码逻辑不再变，就可以不用写测试了。

当然，实际经验起决定作用。直到发布一个缺陷时，TDD都能很好地工作。但是，一旦选择

TDD，还是要提醒自己采取更小、更安全的步伐。

既然选择了TDD，那就来完成转换表吧！

c2/20/Soundex.h

```
std::string encodedDigit(char letter) const {
   const std::unordered_map<char, std::string> encodings {
➤     {'b', "1"}, {'f', "1"}, {'p', "1"}, {'v', "1"},
➤     {'c', "2"}, {'g', "2"}, {'j', "2"}, {'k', "2"}, {'q', "2"},
➤                 {'s', "2"}, {'x', "2"}, {'z', "2"},
➤     {'d', "3"}, {'t', "3"},
➤     {'l', "4"},
➤     {'m', "5"}, {'n', "5"},
➤     {'r', "6"}
   };
   return encodings.find(letter)->second;
}
```

那测试呢？ReplacesConsonantsWithAppropriateDigits中需要3个断言吗？为了回答这个问题，先自问额外的测试是否能增加对此功能特性的理解。答案应该是不能。所以，去掉两个断言，将剩下的断言改用ASSERT_THAT，并选择一个不同的辅音来编码，以此增加我们的信心。

c2/20/SoundexTest.cpp

```
TEST_F(SoundexEncoding, ReplacesConsonantsWithAppropriateDigits) {
➤  ASSERT_THAT(soundex.encode("Ax"), Eq("A200"));
}
```

2.9　如果出现别的情况呢

目前的实现假定能够在encodings映射中找到传入encodedDigit()的字母。之前是为了能慢慢向前推进开发，故作此假设，因为这样就可以写最少的代码来通过每个测试。但是，依然要思考可能需要写的其他代码。

有没有可能传入encodedDigit()的字母没有出现在查找映射中？如果可能，该怎么应对？维基百科并不能回答这个问题。我们可以猜，但是最好的答案来自客户。如果没有客户，可以在网上搜索一下，搜索会马上给出一堆Soundex应用。输入A#会得到A000。答案有了：需要忽略不能识别的字母。

在TDD中，我们可以记下这个既定测试的名称，或者现在就动手写这个测试。有几次我发现提前写一些例外情形的测试，会节省后来调试的时间。所以，现在就来写这个测试吧！

c2/21/SoundexTest.cpp

```
TEST_F(SoundexEncoding, IgnoresNonAlphabetics) {
   ASSERT_THAT(soundex.encode("A#"), Eq("A000"));
}
```

运行这个测试，你会发现测试不是失败了，而是崩溃了。find()返回了指向end()的迭代器，而我们却尝试解引用。当遇到这种情形，先修改一下encodedDigit()，返回一个空字符串。

c2/21/Soundex.h

```
std::string encodedDigit(char letter) const {
   const std::unordered_map<char, std::string> encodings {
      {'b', "1"}, {'f', "1"}, {'p', "1"}, {'v', "1"},
      // ...
   };
➤  auto it = encodings.find(letter);
➤  return it == encodings.end() ? "" : it->second;
}
```

2.10 一次只做一件事

我们需要测试驱动开发出代码用以转换一个词末尾剩下的字母。

c2/22/SoundexTest.cpp

```
TEST_F(SoundexEncoding, ReplacesMultipleConsonantsWithDigits) {
   ASSERT_THAT(soundex.encode("Acdl"), Eq("A234"));
}
```

一个简单的方法是：除了第一个字母，遍历剩下的字母并转换。但是，目前的代码结构还不太容易支持这么做。先重构下代码吧！

但是记住，一次只做一件事。在测试驱动开发时，要保持每一步都不同。在写测试时，不要跑去重构。同样，在尝试让测试通过时也不要重构。将两件事并在一起做时，一旦出岔子会浪费时间。而且这肯定会发生的。

先把刚才写的测试注释掉，暂时停止这个测试。（在Google Mock中，在测试名称前加DISABLE_前缀会跳过执行测试，参见 3.7节，了解这样做的意图。）

c2/23/SoundexTest.cpp

```
➤ TEST_F(SoundexEncoding, DISABLED_ReplacesMultipleConsonantsWithDigits) {
     ASSERT_THAT(soundex.encode("Acdl"), Eq("A234"));
  }
```

我们先专注于重构，修改一下目前的解决方案。不要将整个词传入encodedDigits()，而是将词尾（除了第一个字母外的其余字母）传入encodedDigits()。这样做会让循环处理待转换字母的代码更简洁。同时，使用empty()和front()函数有助于澄清代码的意图。

c2/23/Soundex.h

```
   std::string encode(const std::string& word) const {
➤     return zeroPad(head(word) + encodedDigits(tail(word)));
   }
```

```
private:
   // ...
➤  std::string tail(const std::string& word) const {
➤     return word.substr(1);
➤  }

   std::string encodedDigits(const std::string& word) const {
➤     if (word.empty()) return "";
➤     return encodedDigit(word.front());
   }
```

完成后，运行一下所有的测试，以确保改动不会破坏其他测试。至此，重构工作可以告一段落了。

回到TDD周期的开始，重新启用ReplacesMultipleConsonantsWithDigits，使其失败。可以使用一个基于范围（range）的`for`循环来遍历词尾，让测试通过。

c2/24/Soundex.h

```
std::string encodedDigits(const std::string& word) const {
   if (word.empty()) return "";

➤   std::string encoding;
➤   for (auto letter: word) encoding += encodedDigit(letter);
➤   return encoding;
}
```

既然在`encodedDigits()`中加入了循环，就不用应对传入空字符串之类的防御语句了。在重构环节中，移除它。

c2/25/Soundex.h

```
std::string encodedDigits(const std::string& word) const {
   std::string encoding;
   for (auto letter: word) encoding += encodedDigit(letter);
   return encoding;
}
```

重新运行测试，测试通过！删掉不必要的代码是非常令人满意的，但前提是要有足够的信心。有了测试的保障，做这些代码清理时你会感觉真的很棒！

2.11　限制长度

规则4声明Soundex编码结果必须是4个字符。下面为此写个新的测试。

c2/26/SoundexTest.cpp

```
TEST_F(SoundexEncoding, LimitsLengthToFourCharacters) {
   ASSERT_THAT(soundex.encode("Dcdlb").length(), Eq(4u));
}
```

当用Google Mock运行这个新测试时，会抛出异常。不要担心，因为我们的测试工具捕获了这个异常，生成了测试失败的报告，然后继续运行其他的测试。

```
[ RUN      ] SoundexEncoding.LimitsLengthToFourCharacters
unknown file: Failure
C++ exception with description "basic_string::_S_create" thrown in the test body.
[  FAILED  ] SoundexEncoding.LimitsLengthToFourCharacters (1 ms)
```


默认情况下，Google Mock会记下问题，继续运行剩余的测试。如果你喜欢一遇到未处理的异常就让测试崩溃，可以用下面的命令行选项运行Google Mock：

```
--gtest_catch_exceptions=0
```

gdb（或类似的调试工具）中的栈回溯显示问题出在zeroPad()中。搜索引擎给出的结果显示，创建一个超出最大长度的字符串，导致了_S_create错误。基于这两点事实，我们集中看一下zeroPad()中的字符串构造过程。当编码的长度超出MaxCodeLength时，zeroNeeded溢出了一个值，这使字符串构造函数失败。

TDD方法学提倡的增量方法，使解决问题变得更加容易，因为一旦问题出现，就会暴露出来。不需要调试来精确定位问题，看一眼栈回溯就足够了。

（但是，只要程序出现崩溃，就得思考一下我们的方法。怎么更好地进行测试驱动开发，让问题的根源更加明显？在写zeroPad()时，我们或许想到将其声明为公有的实用方法。那样的话，我们或许会更全面地测试这个方法，以便让其他开发者知道怎么使用它。我们也很有可能会考虑防止zeroPad()创建不合法长度的字符串。）

此问题的解决方案是，修复zeroPad()中的问题。也可以改变encodedDigits()，使之在得到足够的字母时，停止编码。我们选择后者：一旦encoding得到编码，就跳出循环。

c2/26/Soundex.h

```
std::string encodedDigits(const std::string& word) const {
   std::string encoding;
   for (auto letter: word)
➤  {
➤     if (encoding.length() == MaxCodeLength - 1) break;
      encoding += encodedDigit(letter);
➤  }
   return encoding;
}
```

新加的代码不能清晰直接地表达其意图。先把它提取为意图明确的函数isComplete()。

c2/27/Soundex.h

```
std::string encodedDigits(const std::string& word) const {
   std::string encoding;
   for (auto letter: word) {
➤     if (isComplete(encoding)) break;
     encoding += encodedDigit(letter);
```

```
    }
    return encoding;

}

➤ bool isComplete (const std::string& encoding) const {
➤    return encoding.length() == MaxCodeLength - 1;
➤ }
```

2.12　丢掉元音

规则1说要丢掉所有的元音以及w、h和y。 由于想不到更好的名字，先称这些字母为元音类字母。

c2/28/SoundexTest.cpp

```
TEST_F(SoundexEncoding, IgnoresVowelLikeLetters) {
   ASSERT_THAT(soundex.encode("Baeiouhycdl"), Eq("B234"));
}
```

运行一下测试，发现还没添加产品代码，测试就通过了。这是由于`encodedDigit()`对于转换表中找不到的字母会返回空字符串。这样，所有的元音就被编码为空字符串了，并且追加到结果字符串。

如果没有改动类定义，测试就通过了，那背后肯定另有故事（参见3.5节）。可以问自己一个问题："我在测试中做了与期望不一致的事情吗？"

如果接下来的测试也都继续通过，那么应该考虑回滚掉代码改动。测试提前通过的原因，可能是你的步伐有点大，但这样你可能不会感受到TDD带来的好处。在我们的示例中，本来可以先写一个测试，演示该怎么应对不能识别的字母，这样就可以选择返回相同的字母，而非空字符串。

2.13　让测试自我澄清

下一个测试是要处理两个相邻字母有相同数字编码的情形。按照规则3，将用一个数字表示这些字母。这条规则也适用于第一个字母。我们先处理前一种情形，然后再考虑后者。

c2/29/SoundexTest.cpp

```
TEST_F(SoundexEncoding, CombinesDuplicateEncodings) {
   ASSERT_THAT(soundex.encode("Abfcgdt"), Eq("A123"));
}
```

这个测试有点让人迷糊！为了理解为什么Abfcgdt编码为A123，我们需要知道b和f都编码为1，c和g编码为2，d和t编码为3。也可以通过阅读其他测试（如ReplacesConsonantsWithAppropriateDigits）来了解这一事实，但或许应该让测试更加直接。

我们添加一些前置条件（precondition）断言，以帮助阅读代码的人建立这种关联。

c2/30/SoundexTest.cpp

```
TEST_F(SoundexEncoding, CombinesDuplicateEncodings) {

➤    ASSERT_THAT(soundex.encodedDigit('b'), Eq(soundex.encodedDigit('f')));
➤    ASSERT_THAT(soundex.encodedDigit('c'), Eq(soundex.encodedDigit('g')));
➤    ASSERT_THAT(soundex.encodedDigit('d'), Eq(soundex.encodedDigit('t')));

     ASSERT_THAT(soundex.encode("Abfcgdt"), Eq("A123"));
}
```

前件断言不能通过编译，因为encodeDigit()是私有成员函数。先简单把它声明为公有的成员函数。

c2/30/Soundex.h

```
➤ public:
     std::string encodedDigit(char letter) const {
        // ...
     }

➤  private:
      // ...
```

我感觉到读者的惊愕了。

> **提问**：不，等一等！你不能把私有函数改成公有的。
>
> **回答**：我们也有其他的解决方案。可以把测试代码作为Soundex类的友员函数，但是友员往往是一个糟糕的选择，这一点在TDD中也是一样的。可以把这个函数写到另一个类中，譬如SoundexDigitEncoder，但是这似乎过度设计了。也可以不要前置条件断言，而是寻找其他的方法让测试更具可读性。
>
> **提问**：我们一直被教导不要暴露内部实现细节。你不是也应该遵守此金科玉律吗？
>
> **回答**：首先，我们不会随意暴露所有的实现，而只是我们需要的。其次，我们没有暴露过多的实现细节以至于影响到Soundex类的公共接口。是的，我们添加了一个产品客户不需要的函数，但是测试需要。所以，滥用的风险是比较低的，换而得之的是，对于以后必须阅读测试的开发者来说，节省了很多时间。

可以在其他写好的测试中使用前置条件断言。但是，尽量不要这样做。通常，使用名称有意义的常量或局部变量是一个简单有效的方案。此外，试图加一个前置条件断言，可能表示你错过了另一个测试。对吧？可以试着添加一个测试，然后看看它是否消去了对前置条件断言的需要。

为了让测试CombinesDuplicateEncodings通过，可以引入一个局部变量，记录最后一个追加的数字，并在每次循环迭代时更新它。但是光一个局部变量似乎模棱两可。我们还是以一个意图明确的声明开始。

c2/31/Soundex.h

```
std::string encodedDigits(const std::string& word) const {
```

```
   std::string encoding;
   for (auto letter: word) {
     if (isComplete(encoding)) break;
➤    if (encodedDigit(letter) != lastDigit(encoding))
        encoding += encodedDigit(letter);
   }
   return encoding;
}
```

我们知道lastDigit()需要做什么。稍微想一想就能找到一个实现的方法。如下：

c2/31/Soundex.h

```
std::string lastDigit(const std::string& encoding) const {
   if (encoding.empty()) return "";
   return std::string(1, encoding.back());
}
```

2.14 跳出条条框框来测试

现在考虑写下一个测试，其中第二个字母和第一个字母重复。嗯……目前所有测试都是以一个大写字母开始，其余字母是小写，但是这个算法应该是大小写无关的。让我们稍微停一下，先实现一些考虑大小写的测试。（也可以把这个问题加入到测试列表，以后再处理。）

没有指导怎么去处理大小写的规范，但是TDD的好处之一就是，我们可以思考目前手头之外的事情。创建一个健壮的应用程序需要我们能够处理没有明确指明的关键要素。（提示：可以问问你的客户。）

为了能够快速、简单地比较，Soundex算法将类似的词编码至相同的代码。字母的大小写并不影响发音。但为了简化比较Soundex编码，我们将自始至终使用一样的写法。

c2/32/SoundexTest.cpp

```
TEST_F(SoundexEncoding, UppercasesFirstLetter) {
   ASSERT_THAT(soundex.encode("abcd"), StartsWith("A"));
}
```

修改一下`encode()`中的核心算法，将首字母大写，这里我们期待只有一个字母。（`upperFront()`中的映射操作避免了处理字符串结束符带来的潜在问题。）

c2/32/Soundex.h

```
std::string encode(const std::string& word) const {
➤  return zeroPad(upperFront(head(word)) + encodedDigits(tail(word)));
}

➤ std::string upperFront(const std::string& string) const {
➤   return std::string(1,
➤        std::toupper(static_cast<unsigned char>(string.front())));
}
```

有了对大小写的处理，也促使我们去修改测试IgnoresVowelLikeLetters。如上所述，我们期望

算法代码可以忽略大小写的元音字母。但是，我们还是想确认一下。为此，更新测试，验证我们的想法。这样看来，我们从TDD跳转到了事后测试的方式。

c2/33/SoundexTest.cpp

```
TEST_F(SoundexEncoding, IgnoresVowelLikeLetters) {
➤  ASSERT_THAT(soundex.encode("BaAeEiIoOuUhHyYcdl"), Eq("B234"));
}
```

测试通过了，我们可以丢掉刚才更新了的测试。但是为了其他开发者考虑，我们决定保留此测试，用来显式地文档化此行为。

由于不是很确定代码的行为，我们不得不再写一个测试。但是当它立马通过时，我们觉得有必要去审视一下代码。encodedDigits()中的代码有点隐晦和难以理解。我们必须深入考虑来发现以下几点：

- 许多字母没有对应的编码；
- encodedDigit()对于上述字母会返回空字符串；
- 将一个空字符串和encodedDigits()中的变量encodings连接起来没有任何意义。

我们重构一下，以便代码更加明了。首先，对于一个字母，如果encodings表中没有对应的编码，encodedDigit()将返回一个名为NotADigit的常量。然后，在encodedDigits()中增加一个条件表达式，显式地指明NotADigit将被忽略。同时，我们也在lastDigit()中使用这一常量。

c2/34/Soundex.h

```
➤ const std::string NotADigit{"*"};

  std::string encodedDigits(const std::string& word) const {
     std::string encoding;
     for (auto letter: word) {
        if (isComplete(encoding)) break;

➤       auto digit = encodedDigit(letter);
➤       if (digit != NotADigit && digit != lastDigit(encoding))
➤          encoding += digit;
     }
     return encoding;
  }

  std::string lastDigit(const std::string& encoding) const {
➤    if (encoding.empty()) return NotADigit;
     return std::string(1, encoding.back());
  }
  // ...
  std::string encodedDigit(char letter) const {
     const std::unordered_map<char, std::string> encodings {
        {'b', "1"}, {'f', "1"}, {'p', "1"}, {'v', "1"},
        // ...
     };
```

```
    auto it = encodings.find(letter);
➤   return it == encodings.end() ? NotADigit : it->second;
}
```

（上述代码列出了一些增量重构所做的改动，每次改动都经测试验证通过。换句话说，我们并不是一次完成这些改动的。）

让我们继续看一个处理辅音大小写的测试吧！

c2/35/SoundexTest.cpp

```
TEST_F(SoundexEncoding, IgnoresCaseWhenEncodingConsonants) {
   ASSERT_THAT(soundex.encode("BCDL"), Eq(soundex.encode("Bcdl")));
}
```

这里的断言和之前的稍微不同。它声明了对"BCDL"和"Bcdl"的编码结果是一样的。也就是说，我们并不关心实际的编码是什么，只要大写的输入和小写的输入得到的结果一样就行。

我们的解决方案是在查询encodings表时，将字母全部转为小写（在encodedDigit()中）。

c2/35/Soundex.h

```
   std::string encodedDigit(char letter) const {
      const std::unordered_map<char, std::string> encodings {
         {'b', "1"}, {'f', "1"}, {'p', "1"}, {'v', "1"},
         // ...
      };
➤     auto it = encodings.find(lower(letter));
      return it == encodings.end() ? NotADigit : it->second;
   }

private:
➤  char lower(char c) const {
➤     return std::tolower(static_cast<unsigned char>(c));
➤  }
```

2.15　言归正传

在开始前一小节时，我们曾尝试写一个测试来处理第二个字母和第一个字母一样的情形。这又促使我们先将算法改为与大小写无关的。现在，可以回到最初的目标，继续写这个测试了。

c2/36/SoundexTest.cpp

```
TEST_F(SoundexEncoding, CombinesDuplicateCodesWhen2ndLetterDuplicates1st) {
   ASSERT_THAT(soundex.encode("Bbcd"), Eq("B230"));
}
```

我们的解决方案会对encode()的总体策略稍作修改。将整个词传进encodedDigits()进行编码，这样就可以比较第一个字母和第二个字母的编码了。我们只将编码的尾部追加到最终的编码。

在encodedDigits()内部，先将单词的第一个字母编码，这样后续的编码可以与之作比较。

由于encodedDigits()现在是编码整个单词，所以我们修改isComplete()以多容纳一个字母。同时，还修改encodedDigits()中的循环，使之遍历整个单词的词尾。

c2/36/Soundex.h

```
std::string encode(const std::string& word) const {
➤   return zeroPad(upperFront(head(word)) + tail(encodedDigits(word)));
}

std::string encodedDigits(const std::string& word) const {
   std::string encoding;

➤   encoding += encodedDigit(word.front());

➤   for (auto letter: tail(word)) {
      if (isComplete(encoding)) break;

      auto digit = encodedDigit(letter);
      if (digit != NotADigit && digit != lastDigit(encoding))
         encoding += digit;
   }
   return encoding;
}

bool isComplete (const std::string& encoding) const {
➤   return encoding.length() == MaxCodeLength;
}
```


2.16　重构至单一责任的函数

函数encodedDigits()变得越来越复杂。为了将相关的语句分组，我们在其间插入了一些空行。这也揭示了此函数做的事情太多了。

单一责任原则（Single Responsibility Principle，SRP）告诉我们，每个函数的改动都是基于一个原因①。encodedDigits()是违反这一原则的典型例子：它把高层的策略与底层的实现细节糅合到一起了。

encodedDigits()通过两个步骤的算法完成其目的。它首先将首字母的编码追加至变量encoding中，然后遍历剩下的字母，追加结果至encoding。问题是，encodedDigits()中还包含了达成这两步的一些底层实现细节。此函数违背了单一责任原则，因为有两个原因会导致要修改它：想要改变实现细节，或需要改变整个编码策略。

可以将encodedDigits()中的两个步骤提取成两个单独的函数，每个函数各自包含一个抽象概念的实现细节。如此，encodedDigits()中的代码只是声明了解决方案的策略。

① 参见《敏捷软件开发：原则、模式与实践》。

c2/37/Soundex.h

```
std::string encodedDigits(const std::string& word) const {
   std::string encoding;
   encodeHead(encoding, word);
   encodeTail(encoding, word);
   return encoding;
}

void encodeHead(std::string& encoding, const std::string& word) const {
   encoding += encodedDigit(word.front());
}

void encodeTail(std::string& encoding, const std::string& word) const {
   for (auto letter: tail(word)) {
      if (isComplete(encoding)) break;

      auto digit = encodedDigit(letter);
      if (digit != NotADigit && digit != lastDigit(encoding))
         encoding += digit;
   }
}
```

这样看起来好多了。再往前迈进一步，将`encodeTail()`中的`for`循环体提取出来。

c2/38/Soundex.h

```
void encodeTail(std::string& encoding, const std::string& word) const {
   for (auto letter: tail(word))
      if (!isComplete(encoding))
         encodeLetter(encoding, letter);
}

void encodeLetter(std::string& encoding, char letter) const {
   auto digit = encodedDigit(letter);
   if (digit != NotADigit && digit != lastDigit(encoding))
      encoding += digit;
}
```

从视觉上看，重构后的代码还有进一步提升的空间。对单个字母编码的`encodeHead()`函数是不是`encodeTail()`中编码的一个特例呢？尽管去试验吧！因为有了测试的保证，所以你所做的事情将是安全的。目前而言，我们觉得算法的实现已经足够清楚，所以继续吧！

2.17　收尾工作

那元音怎么办呢？规则3说被一个元音（不是h或w）分开的相同编码，应该编码两次。

c2/39/SoundexTest.cpp

```
TEST_F(SoundexEncoding, DoesNotCombineDuplicateEncodingsSeparatedByVowels) {
   ASSERT_THAT(soundex.encode("Jbob"), Eq("J110"));
}
```

再一次通过声明要完成的目标来解决问题。我们修改了encodeLetter()中的条件表达式，在不是重复编码或最后一个字母是元音的情况下，追加一个数字。这个声明也敦促了一些其他的相应改动。

c2/39/Soundex.h

```
   void encodeTail(std::string& encoding, const std::string& word) const {
➤     for (auto i = 1u; i < word.length(); i++)
➤        if (!isComplete(encoding))
➤           encodeLetter(encoding, word[i], word[i - 1]);
   }

➤  void encodeLetter(std::string& encoding, char letter, char lastLetter) const {
      auto digit = encodedDigit(letter);
      if (digit != NotADigit &&
➤           (digit != lastDigit(encoding) || isVowel(lastLetter)))
         encoding += digit;
   }

➤   bool isVowel(char letter) const {
➤      return
➤         std::string("aeiouy").find(lower(letter)) != std::string::npos;
➤   }
```

把最后一个字母传入isVowel()中是最好的方法吗？不过，这个方法直接且具有表达力。我们暂时先这样做。

2.18　漏了什么测试吗

我们很少拥有所有的规范。很少有人会如此幸运。即便是Soundex的规则，看似完整，实际却不能蕴含所有情形。在编程过程中，一些测试或代码实现经常会激发我们进行其他方面的思考。一般而言，要么把这些思考结果记在脑子里，要么写到一个列表或记事本中。下面就是对Soundex进行思考而得到的列表。

- 若给定的词中含有分隔符，如句点（例如，Mr.Smith），该怎么办？应该忽略它们（就像现在做的这样），抛出一个异常（假定客户应该把词合理地分好），还是做些其他操作？说到异常，怎样以测试驱动的方法在代码中加入异常处理？在4.4.5节中，你将学到怎样设计期望抛出异常的测试。
- 空字符串该怎样编码？（或者说，可以假定不会接收到一个空字符串输入吗？）
- 该怎样处理非英语字母中的辅音（如ñ）？Soundex算法依然适用吗？isVowel()函数需要支持带变音符的元音吗？

许多这样的考虑对于设计一个健壮的Soundex类至关重要。若未能处理好它们，在实际使用中，应用程序可能会失效。

对于诸如此类的问题，我们其实没有确切的答案。作为程序员，我们应该已经学会适时作出自己的决定。但往往更好的方法就是去问客户，甚至可以和他们一起制定验收测试（参见10.3节）。

“系统应该按照A来做，还是按照B来做？”以往而言，我们会直接把选择代码化，然后继续。这样做的后果是，未能及时将我们作出的决定写入文档。有时，解决方案B可能会被夹杂在一大堆其他代码中。当然，可以分析代码来确定我们作出了什么选择，但这通常很耗时。相信我们都曾经花过无数小时试图判定代码的行为吧！

相反，TDD留下了一份清晰的文档。我们可以毫不费力地回忆起数月前作出的决定。

和迄今为止所做的一样，我们可以采用TDD方式为前面提出的几个问题开发解决方案。这些漏掉的测试就留给读者作为练习吧！

2.19　解决方案

我们用测试驱动方法开发出了Soundex的解决方案。这个解决方案绝不是唯一的或最好的，但是我们有足够的信心去发布了（除了2.18小节提及的一些未解问题），这才是最重要的。

我自己对Soundex做了好几次测试驱动开发，每次都得到不同的解决方案。其中大部分解决方案的差异很小，但有一个差异很大（而且工作起来很糟糕），这是由我积极地以极具描述性的方式来解决问题而导致的。每一次Soundex TDD经历都让我更好地了解此算法，与此同时，我也学会了更多可以在TDD中很好地工作的东西。

多次测试驱动开发Soundex算法后，你会发现类似的好处。重复使用TDD方法去开发同一个示例可称为套路（kata）。参见11.5节，以了解更多相关信息。

任何解决方案的实现并非仅有一种正确的方式。下面是一个解决方案应具备的一些重要特征。

- 它实现了客户的需求。如果没有，那么不管怎样，它都不是好的解决方案。在TDD中，你编写的测试能够帮助你了解你的解决方案是不是客户要的。性能可能是众多客户需求中的一项。你的一部分职责就是理解他们的性能需求，如果没必要的话，就不要花费时间去做性能优化。
- 它可以工作。如果一个解决方案有大量的缺陷，那么构建得再优雅，也不是好的解决方案。TDD可以帮助确定我们交付的软件能以期望的方式工作。TDD不是银弹[1]。你交付的软件依然会有缺陷，所以照样需要许多其他方式的测试。但是，TDD会让你发布的代码包含非常少的缺陷。

---

[1] 银弹（Silver Bullet）是欧洲民间传说中用来对付怪物的杀手锏武器，后来银弹经常被比喻为一种非常有效的解决方案。Frederick在1987年的IFIPS会议上发表了题为“No Silver Bullet，Essense and Accidents of Software Engineering”的论文。由于这篇论文引发了剧烈的的争论，后来Frederic在《人月神话》一书中对一些公开的批评做了说明，并更新了论文中的一些观点，有兴趣的读者可以找来看看。——译者注

- 它易于理解。对于编写得不好的代码，每个人都需要花费大量的时间去理解。TDD让你可以安全地重新组织代码以提高可读性。
- 它易于修改。通常，容易修改的代码意味着高质量的设计。TDD使你可以持续地修改，以保持设计的高质量。

我们的解决方案不是过程式的。整个算法不是在一个函数中，不能一次从头读到尾。相反，我们以许多小的成员函数的方式完成整体实现，许多函数只有一两行代码。每个函数的代码实现一个抽象。第一次遇到这样的代码可能会导致中风发作：晕！到底工作是在哪完成的？（要了解为什么以这样的方式实现，参见6.2节。）

2.20 Soundex 类

因为已准备好提交代码，所以让我们来纵览一下整个解决方案。我们觉得还没有足够的理由去单独拆分出实现文件（.cpp），但是这是成为系统产品代码所必需的一步。

c2/40/SoundexTest.cpp

```
#include "gmock/gmock.h"
#include "Soundex.h"
using namespace testing;

class SoundexEncoding: public Test {
public:
   Soundex soundex;
};

TEST_F(SoundexEncoding, RetainsSoleLetterOfOneLetterWord) {
   ASSERT_THAT(soundex.encode("A"), Eq("A000"));
}

TEST_F(SoundexEncoding, PadsWithZerosToEnsureThreeDigits) {
   ASSERT_THAT(soundex.encode("I"), Eq("I000"));
}

TEST_F(SoundexEncoding, ReplacesConsonantsWithAppropriateDigits) {
   ASSERT_THAT(soundex.encode("Ax"), Eq("A200"));
}

TEST_F(SoundexEncoding, IgnoresNonAlphabetics) {
   ASSERT_THAT(soundex.encode("A#"), Eq("A000"));
}

TEST_F(SoundexEncoding, ReplacesMultipleConsonantsWithDigits) {
   ASSERT_THAT(soundex.encode("Acdl"), Eq("A234"));
}

TEST_F(SoundexEncoding, LimitsLengthToFourCharacters) {
   ASSERT_THAT(soundex.encode("Dcdlb").length(), Eq(4u));
```

```
}

TEST_F(SoundexEncoding, IgnoresVowelLikeLetters) {
   ASSERT_THAT(soundex.encode("BaAeEiIoOuUhHyYcdl"), Eq("B234"));
}

TEST_F(SoundexEncoding, CombinesDuplicateEncodings) {
   ASSERT_THAT(soundex.encodedDigit('b'), Eq(soundex.encodedDigit('f')));
   ASSERT_THAT(soundex.encodedDigit('c'), Eq(soundex.encodedDigit('g')));
   ASSERT_THAT(soundex.encodedDigit('d'), Eq(soundex.encodedDigit('t')));

   ASSERT_THAT(soundex.encode("Abfcgdt"), Eq("A123"));
}

TEST_F(SoundexEncoding, UppercasesFirstLetter) {
   ASSERT_THAT(soundex.encode("abcd"), StartsWith("A"));
}

TEST_F(SoundexEncoding, IgnoresCaseWhenEncodingConsonants) {
   ASSERT_THAT(soundex.encode("BCDL"), Eq(soundex.encode("Bcdl")));
}

TEST_F(SoundexEncoding, CombinesDuplicateCodesWhen2ndLetterDuplicates1st) {
   ASSERT_THAT(soundex.encode("Bbcd"), Eq("B230"));
}

TEST_F(SoundexEncoding, DoesNotCombineDuplicateEncodingsSeparatedByVowels) {
   ASSERT_THAT(soundex.encode("Jbob"), Eq("J110"));
}
```

c2/40/Soundex.h

```
#ifndef Soundex_h
#define Soundex_h

#include <string>
#include <unordered_map>

#include "CharUtil.h"
#include "StringUtil.h"

class Soundex
{
public:
   static const size_t MaxCodeLength{4};

   std::string encode(const std::string& word) const {
      return stringutil::zeroPad(
         stringutil::upperFront(stringutil::head(word)) +
            stringutil::tail(encodedDigits(word)),
         MaxCodeLength);
   }

   std::string encodedDigit(char letter) const {
```

```
      const std::unordered_map<char, std::string> encodings {
         {'b', "1"}, {'f', "1"}, {'p', "1"}, {'v', "1"},
         {'c', "2"}, {'g', "2"}, {'j', "2"}, {'k', "2"}, {'q', "2"},
                     {'s', "2"}, {'x', "2"}, {'z', "2"},
         {'d', "3"}, {'t', "3"},
         {'l', "4"},
         {'m', "5"}, {'n', "5"},
         {'r', "6"}
      };
      auto it = encodings.find(charutil::lower(letter));
      return it == encodings.end() ? NotADigit : it->second;
   }

private:
   const std::string NotADigit{"*"};

   std::string encodedDigits(const std::string& word) const {
      std::string encoding;
      encodeHead(encoding, word);
      encodeTail(encoding, word);
      return encoding;
   }

   void encodeHead(std::string& encoding, const std::string& word) const {
      encoding += encodedDigit(word.front());
   }

   void encodeTail(std::string& encoding, const std::string& word) const {
      for (auto i = 1u; i < word.length(); i++)
         if (!isComplete(encoding))
            encodeLetter(encoding, word[i], word[i - 1]);
   }
   void encodeLetter(std::string& encoding, char letter, char lastLetter) const {
      auto digit = encodedDigit(letter);
      if (digit != NotADigit &&
            (digit != lastDigit(encoding) || charutil::isVowel(lastLetter)))
         encoding += digit;
   }

   std::string lastDigit(const std::string& encoding) const {
      if (encoding.empty()) return NotADigit;
      return std::string(1, encoding.back());
   }

   bool isComplete(const std::string& encoding) const {
      return encoding.length() == MaxCodeLength;
   }
};

#endif
```

等等，有些东西已经变了！`head()`、`tail()`和`zeroPad()`在哪？`isVowel()`和`upper()`呢？`lastDigit()`看起来也不同了！

哈哈！当你忙于阅读本书的时候，我做了额外的一些重构工作。这些消失的函数（原先定义在Soundex中）现在被作为自由函数声明在`StringUtil.h`和`CharUtil.h`中。通过这个小的重构，它们成了高度可重用的函数。

简单地将这些函数从Soundex中移除还不够。作为公用的工具函数，它们需要恰当的描述，以便其他程序员能够理解它们的意图和用法。这也就意味着，需要一些测试告诉程序员如何在代码中使用这些函数。可以在本书的源代码中找到这些工具函数及它们的测试。

我们用测试驱动方法开发出了Soundex的一种解决方案。这个方案的演化完全取决于你。你越多地践行TDD，你的解决方案风格也就演变得越多。两年前测试驱动开发出的结果和我今天测试开发出的结果有着天壤之别。

2.21　结束语

本章中，你亲身经历了一个实际的TDD过程。你是自己编写代码的吗？如果是，你已经实现了一个几乎可以在产品代码中使用的Soundex类。如果不是，打开编辑器去实现Soundex类吧！通过写代码学习比简单阅读代码更有效。

准备好真正去学TDD了吗？再次出发！再一次测试驱动开发Soundex类，但这次不要阅读本章（除非遇到困难）。以不同顺序实现Soundex规则会发生什么呢？采用不同的编程风格又会怎样呢？如果每个规则都用一个函数变换输入，然后把它们串起来作为Soundex的实现，结果又会如何呢？

如果你准备好继续学习了，下一章会以整体的视角审视TDD。它介绍了TDD的一些基本定义，并提供了为取得成功所需要的战略和战术性建议。

第 3 章

测试驱动开发基础

3.1 开场白

上一章中的详实示例展示了TDD在实践中的样子。在完成示例的过程中，我们接触到了很多概念和良好的实践准则。（如果你已经熟悉TDD，或许跳过了上一章。）本章将深入讨论这些主题，介绍更多的背景知识及其背后的因果关系。

- 单元的定义
- TDD的周期：红–绿–重构
- TDD的三条准则
- 为什么不要忽略测试失败
- 成功的思维
- 成功的方法

在学习过上述主题后，你将在TDD流程和概念方面打下坚实的基础。

3.2 单元测试和测试驱动开发基础知识

TDD会产出单元测试。单元测试验证了一个代码单元的行为，这里的代码单元是一个应用中最小的、可测的一段代码。通常而言，开发单元测试和代码单元要使用一致的编程语言。

3.2.1 单元测试的组织和执行

单元测试包括一个描述性的名称和一系列代码声明，从概念上可以细分成四个（有序的）部分：

(1) 设置能够运行上下文的语句，这一部分是可选的；
(2) 一条或多条能构成你想要验证的行为的语句；
(3) 一条或多条验证期望输出的语句；
(4) 清理工作的语句（例如，释放所分配的内存），这一部分是可选的。

有些开发者把前三部分称为Given-When-Then。换言之就是给定一个上下文，执行测试，继而验证行为。还有一些开发者称之为Arrange-Act-Assert（参见第4章）。

一般来说，我们会将关联的测试组织到一个源文件中。和大多数单元测试工具一样，使用Google Mock可以将单元测试按逻辑分组到fixture中。你可以给每个TDD的类配一个fixture，但是也不要拘泥于此形式。一个C++类对应多个fixture，或一个fixture测试多个类中相关联的行为，这都是很常见的。参见第4章了解更多的信息。

可以选择众多可用的C++单元测试工具中的一个来执行单元测试。尽管这些工具大部分都很相似，但并没有一个通用的标准。所以，不能直接在各个工具间移植C++单元测试。每个工具都定义了不同的规则，诸如组织测试的方式、断言的形式，以及执行测试的方式。

我们把执行一遍所有的测试称为*运行测试*（test run或suite run）。在运行测试的过程中，工具会枚举所有的测试，独立地执行每个测试。对于每个测试来说，测试工具都是从头到尾执行其中的语句的。当执行一条断言语句时，如果断言所期待的条件不满足，那么测试就会失败。反之，测试通过。

单元测试实践方式多种多样，测试的粒度也各异。大多数开发者编写单元测试（但不一定在使用TDD）仅仅为了使用工具提供的验证功能。通常这些开发者在完成产品的一个功能部分之后，就会去编写相应的测试。由于开发人员没有以测试的思维去写测试，因此很难去编写和维护这些后于开发的测试。而我们将使用TDD，并且会做得更好。

3.2.2　测试驱动单元

与plain ol' unit testing（POUT）[①]不同，TDD是一个定义更加简明的流程，并且也使用了单元测试。但在TDD中，我们会先写测试，并且保持测试的粒度尽量小且一致，这样做会有许多好处，其中最重要的就是可以安全地修改现有代码。

在TDD中，你是以非常小的步伐，增量地往系统中添加新行为。换句话说，为了往系统中加入新的行为，首先你会去写一个测试来定义这个行为。这个起初运行失败的测试将驱使你去实现相应的行为。

"非常小？"事实上并没有一个标准的大小限制，所以你需要采用一个适当的大小。每个测试应当代表你能想到的最小的、有意义的增量。测试最好包含一到三行代码（参见4.2.4节）和一个断言（参见7.3节）。当然，也会有许多代码比较多的测试，这没关系，但坚持"适当"的原则会提醒你反过来思考测试的大小，即这些测试是不是做得太多了？

仅需几分钟就可以写好包含三行以内语句及一个断言语句的测试，同时仅仅需要几分钟就可实现对应的行为，一个断言可以验证的代码有多少呢？这些为运行失败测试而加入到系统中的相

① 这里指的是在代码开发完成后再编写测试的情况。——译者注

关小代码段被称为*逻辑分组*，或*代码单元*。

不要写覆盖大量功能的测试。应将此类测试放在别处，或许作为验收测试（客户测试）或系统测试。本书把这种为集成代码准备的测试称作*集成测试*（参见10.3节）。集成测试验证的代码必须与其他代码或外部实体（文件系统、数据库、网络协议和其他API）集成。相反，你可以使用单元测试独立地验证代码单元。

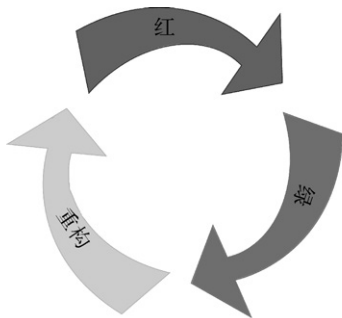
我们可以从TDD（参见3.4节）的第一条原则得知，只在让失败测试通过时才编写产品代码。为了遵循这条原则，你就不可避免地需要开发与外部实体交互的代码。这样做又把你引向集成测试，但是没关系。TDD并不阻止你这么做。不要太在意术语上的不同。重要的是先指定行为，系统地测试驱动开发系统中的每一块代码。

我们将在第5章学习怎样利用测试替身切断对外部实体的依赖。这样一来，天生执行较慢的集成测试将演化为快速执行的单元测试。

3.3 测试驱动开发周期：红－绿－重构

在做测试驱动开发时，要重复以下简短的周期：

(1) 写一个测试（“红”）；
(2) 让测试通过（“绿”）；
(3) 优化设计（“重构”）。



这个周期通常被称为：红–绿–重构，这来源于TDD中使用的单元测试工具。红（失败）和绿（通过）来源于SUnit（第一个支持TDD的单元测试工具[①]）和类似的GUI工具，这些工具用颜色提供测试结果的快速反馈。Google Mock以文本方式输出，如果终端支持颜色输出，也会使用红和绿表示测试结果。

在重构阶段，要确保代码库有尽可能最好的设计，这让你能够以合理的开销去扩展和维护系

① http://sunit.sourceforge.net

统。当你想在不改变代码行为的情况下改进设计时，则需要重构（这一术语因Matrin Fowler的《重构：改善既有代码的设计》而普及），而有了测试的保障，重构就能安全地进行。参见第6章获取更多信息。

思考和测试驱动开发

将TDD的周期记在脑海里的目的是让你适应两件事。第一，你需要写一个测试说明行为。第二，每个周期都要清理代码。养成这一习惯会使你的思想得到解放，进而去思考在增量开发解决方案中更具挑战性的问题。

在TDD周期中的每一步，你必须能够回答以下问题。

- **写一个小的测试**。怎样才算可以增量开发的最小行为呢？（参见3.7节）系统中已经存在这样的行为了吗？怎样让测试名称准确表达行为？测试中使用的接口是客户端代码使用这一行为的最好方式吗？
- **确保新的测试是失败的**。如果没有失败，为什么？这个行为已经在系统中存在了？你忘记编译了吗？是不是在上个测试中步子迈大了？断言是否有效？
- **写出你认为可以让测试通过的代码**。你写的代码是不是刚好满足测试说明的行为要求？你清楚刚才写的代码中哪些地方需要整理吗？你遵循团队的标准了吗？
- **确保所有测试都能通过**。如果没有，你的编码正确吗？或是你的规范正确吗？
- **整理刚才的代码改动**。怎么做才能让你的代码符合团队标准？新的代码和系统中其他要清除的代码有重复吗？代码有没有坏味？遵循好的设计原则了吗？除了当下要做的设计和代码整理工作，你还知道其他什么？设计是朝好的方向发展吗？你的代码改动会导致需要修改其他地方的代码吗？
- **确保所有测试再次通过**。确信你的单元测试覆盖率够高吗？你是不是应该运行一些速度较慢的测试集合，以便有信心继续前行？下一个测试是什么？

本书中包含大量帮助你解决这些问题的信息。

在一天的开发工作中，你时常需要思考大量的东西。虽然TDD周期很简单，但是构建产品级代码却非易事。践行TDD绝非无意识的练习。

3.4　测试驱动开发的三条准则

Robert C. Martin（Bob大叔）提出了践行TDD的简明规则[1]。

(1) 只在让失败测试通过时才编写产品代码。

(2) 当测试刚好失败时，停止继续编写。编译失败也是失败。

---

[1] http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

(3) 只编写能刚好让一个失败测试通过的产品代码。

规则1是说先写一个测试。用单元测试的方式来理解和说明必须要加入到系统中的行为。

规则2是说尽量增量地前进。你可以在每写一行代码后，继续写下一行代码之前，得到一些反馈信息（通过编译或运行测试）。

我们需要在测试驱动开发RetweetCollection类并写完第一个测试时立刻停下来。

c3/1/RetweetCollectionTest.cpp

```
#include "gmock/gmock.h"

TEST(ARetweetCollection, IsEmptyWhenCreated) {
   RetweetCollection retweets;

}
```

我们还没定义RetweetCollection类，所以我们知道这行测试代码不能编译通过。这时就需要写一些代码来定义这个类，重新编译，直到通过为止。只有到这一步，我们才能去写一个断言来验证集合是空的。

规则2是存有争议的，它还没有作为TDD标准被广泛接受。在C++中，如果编译时间过长，这样做可能会降低效率。你可能觉得先完成整个测试效率会更高。但是，在摒弃此规则前，还是先老老实实地遵守（作为一个TDD的实践者，我会按时所需地遵循此规则）。

规则3说不要写多于测试规定的代码。这也是规则1说“让失败测试通过”的原因。如果你写的代码多于需要的（例如你实现了一个没有测试对应的行为），就违反了规则1，因为你在这之后写的测试可能会立刻运行通过。

有时候你会发现，在掌握测试驱动开发过程中，先写一个失败的测试是非常困难的（下一节将会进一步讨论这个问题）。

学习时遵循这些规则有助于你日后理解一些会破坏它们的潜在因素。

3.5 表里不一

TDD的第一条规则要求在写代码前，先写出一个运行失败的测试。从逻辑上讲，这是一个可以遵守的简单规则。如果你写的代码仅仅能让一个测试通过，那么针对额外功能的测试自然会失败。但实际上，有时你会发现刚刚写完的一个测试立刻就通过了。我将这种非期望事件称为*提前通过*（premature passes）。

下列原因可能会让测试提前通过：

- ❑ 运行了错误的测试
- ❑ 测试了错误的代码

- 不当的测试规范
- 对系统的无效假设
- 不佳的测试顺序
- 相关联的产品代码
- 过度编码
- 确定性测试

3.5.1　运行了错误的测试

当我们静下心来，要做的第一件事就是运行测试集。你有多少测试？每当你新加一个测试，都会焦急地等待测试运行后显示的测试数目。“嗯……为什么我的测试过了？我应该有43个测试，但是现在只运行了42个。这次运行的测试不包含我的新测试。”

跟踪测试数目有助于你快速判断是不是犯了以下愚蠢的错误：

- 运行了错误的测试集；
- Google Mock过滤器忽略了新的测试（参见4.3节）；
- 测试并没有经过编译或链接；
- 测试被禁止了（参见3.7节）。

如果没有第一时间发现测试失败，那么问题会变得更糟。试想你写完一个测试，跳过运行测试集这一环节，继续写代码，然后运行测试。这时你认为测试通过的原因是编写了正确的代码。但是，事实上测试通过的原因是本轮运行并不包含新的测试。而你在发现这一问题前或许还在沾沾自喜呢！

遵循TDD周期能够让你避免因此类错误而浪费许多时间。

3.5.2　测试了错误的代码

和运行了错误的测试一样，你也会测试了错误的代码。通常而言，这和编译错误有关，但也并非总是如此。下面罗列了测试错误代码的原因。

- 忘记了保存或编译。这种情况下，“错误”的代码是上次编译的版本，并没有包含新的改动。为了避免这种错误，你可以让单元测试的运行依赖编译，或者将运行测试作为构建后的一个步骤。
- 构建失败了，但你却没注意到，还沾沾自喜地认为构建成功了。
- 构建脚本有缺陷。你是不是链接到了错误的对象模块？目标对象模块是不是和另外一个对象模块重名了？
- 你在测试错误的类。如果你在使用测试替身做些有趣的事情，它允许你使用多态的替换

以便测试更加容易。而运行的测试使用的实现很可能跟你预想的不一致。

3.5.3 不当的测试规范

你写了个测试，而它断言的东西却不是你要的。

试想我们写了一个测试，如下所示：

```
TEST_F(APortfolio, IsEmptyWhenCreated) {
   ASSERT_THAT(portfolio.isEmpty(), Eq(false));
}
```

3

喔，不，从测试的名称上看，Portfolio类实例在创建时应该为空。所以我们期望的`isEmpty()`返回值应该为`true`，而不是`false`。

如果出现提前通过的情况，要重读一下测试，确保它规定了正确的行为。

3.5.4 对系统的无效假设

试想你写了一个测试，它一运行就通过了。你也确定运行了正确的测试，并且所测的代码也是对的。你重新审阅了测试，确信它的行为是你想要的。这么说来，系统中已经存在了测试所规定的功能。嗯……

之所以写了这个测试，是因为你假定这个行为在系统中是不存在的（参见3.5.8节）。这个通过的测试告诉你之前的假设是错误的，所测行为已经在系统中了。这时，你必须停下来，就之前你认为已经添加的行为来分析系统，直到了解了足够的情况再进行下一步的操作。

在这种情况下，测试通过是好事。重要的事情已经警示过你了。或许你误解了第三方组件的行为。花点时间仔细考查一下也许就可以避免发布一个缺陷。

3.5.5 不佳的测试顺序

RetweetCollection类接口需要`size()`和一个便捷成员函数`isEmpty()`。我们会在这两个接口的测试通过后，重构`isEmpty()`实现，并将任务委派给`size()`，这样就不需要为两个相关的概念提供不同的算法。

c3/2/RetweetCollectionTest.cpp

```
#include "gmock/gmock.h"
#include "RetweetCollection.h"

using namespace ::testing;

class ARetweetCollection: public Test {
public:
```

```
   RetweetCollection collection;
};

TEST_F(ARetweetCollection, IsEmptyWhenCreated) {
   ASSERT_TRUE(collection.isEmpty());
}

TEST_F(ARetweetCollection, HasSizeZeroWhenCreated) {
   ASSERT_THAT(collection.size(), Eq(0u));
}
```

c3/2/RetweetCollection.h

```
#ifndef RetweetCollection_h
#define RetweetCollection_h
class RetweetCollection {
public:
   bool isEmpty() const {
      return 0 == size();
   }

   unsigned int size() const {
      return 0;
   }
};
#endif
```

为了扩展空的概念，我们写了下面的测试，以确保一旦将tweets加入到retweet集合中，集合就不为空。

c3/3/RetweetCollection.h

```
#include "Tweet.h"

TEST_F(ARetweetCollection, IsNoLongerEmptyAfterTweetAdded) {
   collection.add(Tweet());

   ASSERT_FALSE(collection.isEmpty());
}
```

（但目前为止，我们不必关心tweet的内容，只需为`Tweet.h`中的Tweet类定义提供`class Tweet{};`即可。）

这里只要引入一个跟踪集合大小的变量，就可以让测试IsNoLongerEmptyAfterTweetAdded通过。

c3/3/RetweetCollection.h

```
#include "Tweet.h"

class RetweetCollection {
public:
➤   RetweetCollection()
➤      : size_(0) {
```

```
➤ }

     bool isEmpty() const {
        return 0 == size();
     }

     unsigned int size() const {
➤       return size_;
     }

➤    void add(const Tweet& tweet) {
➤       size_ = 1;
➤    }

➤ private:
➤    unsigned int size_;
  };
```

但是，问题来了。如果我们想知道在加入一个tweet后`size()`的行为，它会立刻通过测试。

c3/4/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, HasSizeOfOneAfterTweetAdded) {
   collection.add(Tweet());

   ASSERT_THAT(collection.size(), Eq(1u));
}
```


我们该怎样做才能避免这样的事情发生呢?

在回答这个问题之前，我们或许需要换个不同的视角看待问题。需要写这个测试吗?TDD是为了增强信心的，而不是穷举测试。我们一旦对一个实现有了足够的信心，就可以不用写测试。或许应该删掉测试HasSizeOfOneAfterTweetAdded，然后继续。又或者，应该保留它用作文档之用。

否则，只要遇到测试提前通过，我们就该看看为之前测试的通过所编写的代码。是不是写了过多的代码?就当前情形而言，不是，现在的代码已经简单得不能再简单了。但是如果在引入`size()`前就扩展空的行为了呢?如果测试顺序是：IsEmptyWhenCreated、IsNoLongerEmptyAfter-TweetAdded、HasSizeZeroWhenCreated，情况会稍有不同。

c3/5/RetweetCollection.h

```
class RetweetCollection {
public:
   RetweetCollection()
      : empty_(true) {
   }

   bool isEmpty() const {
      return empty_;
   }
```

```
    void add(const Tweet& tweet) {
       empty_ = false;
    }

    unsigned int size() const {
       return 0;
    }

private:
    bool empty_;
};
```

这时，我们不能把size()和isEmpty()联系起来，因为还没有测试规定size()的行为。所以，也不能重构size()，它依然可以返回一个硬编码的值。如果现在加入HasSizeOfOneAfter-TweetAdded，测试就会失败。让测试通过的最简单的方法似乎有点古怪，但是没关系！

c3/6/RetweetCollection.h

```
unsigned int size() const {
➤   return isEmpty() ? 0 : 1;
}
```

程序员很容易犯懒。你可能不太愿意回滚代码改动，从头来过以便找到一个避免提前通过的路径。但是，你能在此返工的过程中学到一些重要且有价值的东西。如果不这么做的话，至少你也要绞尽脑汁去思考怎样才可能避免提前通过，这或许会帮助你避免再次出现此类情况。

3.5.6　相关联的产品代码

前一节中介绍的isEmpty()是让客户代码更简洁的便捷方法。它和大小是联系在一起的（我们也这样编写代码）。当大小为零时，集合为空；反之，集合不为空。

添加诸如isEmpty()的便捷方法会导致重复接口的出现。它代表客户可以用不同的方式，与已经测试驱动开发出的行为交互。这意味着isEmpty()的测试会自动通过。但是，我们仍然需要说明它的行为。

在向RetweetCollection类中添加新的功能时，例如合并相似的tweet，我们需要验证新的行为是否能合理地影响集合的大小和空。有几种方法可以同时验证这一点。

第一种方法是，对于每一个与大小相关的断言，在判断是否为空的函数上加上第二个断言。这种方法会导致不必要的重复代码，测试也会显得混乱不堪。

c3/7/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, DecreasesSizeAfterRemovingTweet) {
   collection.add(Tweet());

   collection.remove(Tweet());

   ASSERT_THAT(collection.size(), Eq(0u));
```

```
   ASSERT_TRUE(collection.isEmpty()); // 不要这么做
}
```

第二种方法是，对每个针对大小的测试，也为空写个测试。虽然这符合“一个测试一个断言”，但仍会产生大量的重复代码。

c3/8/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, DecreasesSizeAfterRemovingTweet) {
   collection.add(Tweet());
   collection.remove(Tweet());
   ASSERT_THAT(collection.size(), Eq(0u));
}
// 避免这样做
TEST_F(ARetweetCollection, IsEmptyAfterRemovingTweet) {
   collection.add(Tweet());
   collection.remove(Tweet());
   ASSERT_TRUE(collection.isEmpty());
}
```

第三种方法是编写一个辅助方法或自定义断言。大小和空之间有概念上的联系，因此或许应该建立断言方面的概念联系。

c3/9/RetweetCollectionTest.cpp

```
➤ MATCHER_P(HasSize, expected, "") {
➤    return
➤       arg.size() == expected &&
➤       arg.isEmpty() == (0 == expected);
➤ }

TEST_F(ARetweetCollection, DecreasesSizeAfterRemovingTweet) {
   collection.add(Tweet());

   collection.remove(Tweet());

➤    ASSERT_THAT(collection, HasSize(0u));
}
```

Google Mock中的`MATCHER_P`宏自定义了一个匹配器，它接受一个参数。参见https://code.google.com/p/googlemock/wiki/CheatSheet获取更多的信息。如果你使用的单元测试工具不支持自定义匹配器的话，那么可以使用一个简单的帮助方法达到同样的效果。

第四种方法是为这两个概念显式地创建测试，说明它们之间的联系。从某种意义上讲，这是文档化的一种方式。这样，在以后的测试中就不用为测试空担忧了。

c3/10/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, IsEmptyWhenItsSizeIsZero) {
   ASSERT_THAT(collection.size(), Eq(0u));

   ASSERT_TRUE(collection.isEmpty());
}
```

```
TEST_F(ARetweetCollection, IsNotEmptyWhenItsSizeIsNonZero) {
   collection.add(Tweet());
   ASSERT_THAT(collection.size(), Gt(0u));

   ASSERT_FALSE(collection.isEmpty());
}
```

（在表达式Gt(0)中，Gt表示大于）。

在这两个测试中，用于断言大小的是前置条件断言。从技术上讲，它们是不需要的，但在本例中则是为了突显这两个概念之间的联系。

通常，我喜欢第四种方法，虽然有时候第三种方法也很有效。TDD不是精确的科学，你可以使用不同的方法，这意味着你需要不时地停下来评估当前方案，进而选择适用于当前上下文的最佳方案。

3.5.7　过度编码

编写过程序的人往往自认为清楚最终解决方案需要哪些技术。“我知道这里需要使用一个字典数据结构，那么目前就先在map中编码吧。”或者，“是的，我们需要处理代码中抛出的异常，但我们可以先在catch块中记录错误然后重新抛出。”

优秀的程序员往往有这样的倾向和直觉，他们知道代码需要做什么。你不需要摒弃这些自然萌发的想法。我们可以通过了解基于哈希的数据结构来找到更好的解决方案，并且知道它会出现的错误类型也是至关重要的。

如果你想成功运用TDD，就必须确保增量地引入这些想法，并配以相应的测试。保持“红–绿–重构”的节奏能提供许多安全保障，同时也有助于单元测试数目和覆盖率的增长。

提前引入map有时会导致测试提前通过，因为这样做会覆盖代码中需要处理的许多情形。由此带来的后果是，你会忽略大量有用的测试。相反，你可以慢慢地改进底层的数据结构，用一些失败的测试来证明解决方案需要一般化处理。

有时候你会觉得自己甚至不需要一个map。与此同时，不做多余需求的实现可以保持代码简单。这样做也可以避免在存在更简单的解决方案时，引入长时间的过度复杂的代码。

为异常处理编写测试可以帮助客户端程序员更好地理解怎样与你的类交互。这样做也能帮助你清楚地理解并记录有可能出问题的情形。

学会仅仅编写足够的代码是TDD中更加有挑战性的事情之一，但是你一旦掌握，将受益无穷。恪守“红–绿–重构”有助于强化TDD的增量方法。

3.5.8 确定性测试

有时候你并不知道代码在一些特定情形下是怎样工作的。你认为TDD的代码应该是一个完整的解决方案，但并不确定。“我们的算法处理这一情形了吗？”你可以有针对性地写一个测试来探测系统行为。如果测试失败，你就仍处于“红–绿–重构”周期，这个失败的测试将会促使你为新的情形编写代码。

如果测试通过，好极了！系统在按照预期工作。你可以继续开发。但是应该保留还是丢弃这个新测试呢？答案取决于其文档的作用有多大。它有助于以后的客户或开发者理解一些重要的事情吗？它有助于说明另一个测试失败的原因吗？如果是的话，保留测试。否则，移除该测试。

当你写一个测试探测系统行为时，代表着你要验证一个关于系统的假设。也许你认为确定性测试和对系统的无效假设（参见3.5.4节）类似。其实它们的不同点是：每当你写一个测试时，你期望它失败还是通过。如果你期望测试失败，但它却通过了，这属于对系统的无效假设。如果你期望这个测试通过，那么这就是确定性测试。

3.5.9 停下来想一下

测试提前通过的情况应该很少见。但是这种情形却很重要，尤其是在学习TDD时。一旦出现此类情况，可以问自己几个问题：我漏掉了什么？我的步伐是不是大了点儿？我有没有犯愚蠢的错误？如果不这样做会是什么结果呢？和编译警告一样，你始终要去揣摩测试提前通过背后的真相。

3.6 成功运用测试驱动开发的思维

TDD是一个原则，能帮助改善设计质量，但不是验证系统功能的普通方法。持有正确的实践心态是成功运用TDD的基础。下面是在运用TDD时一些有效的思维方式。

3.6.1 增量性

TDD以渐进的方法从无到有地开发一个功能完善的系统。每当往系统中加入一个新的行为单元时，你清楚地知道系统仍会正常工作，因为你为该新行为编写了一个测试，并且以往加入的行为也有对应的测试。你只有在新功能与其他所有功能协同工作时才能继续前进。同样，你也清楚地了解系统的设计目的，因为单元测试描述了加入系统中的行为。

TDD的增量方法和敏捷流程相契合（虽然你可以在任何流程中使用TDD）。敏捷开发定义了短的迭代周期（通常一周或两周），在此期间你定义、构建并发布少量功能。每次迭代都代表了需求级别最高的功能特性。在随后的每次迭代中都可以完全改变优先顺序。敏捷开发者甚至可以提前取消项目。这没关系，因为敏捷就是为最高业务需求而生的。将其结果与传统开发方法（先

花费数月时间做需求分析，然后设计，再编码）对比会发现，后者在完成分析和设计前，没有什么有实际意义的产出，而且通常这样的状态会延续到所有编码都完成。

TDD支持类似于增量思维的小步方法。你可以逐个地处理单元，使用测试来定义和验证它们。在任何时间点上，你都可以停止开发，并且知道你已构建了测试所描述的所有系统行为。任何没有测试描述的行为都没实现，而经测试描述的行为则正确且完整地实现了。

3.6.2　测试行为而非方法

TDD初学者常会犯的一个错误是集中精力去测试成员函数。“我们实现了一个`add()`成员函数。再写一个`TEST(ARetweetCollection, Add)`的测试。”但是，写一个完全覆盖add行为的测试需要考虑多种不同的情形。结果是你必须将许多不同的行为编码进同一个测试。此时，测试就没有文档价值了，同时，理解一个测试花费的时间也会增加。

相反，你要把注意力放在行为或描述行为的情形上。如果加入一个之前已经加入的tweet会发生什么？如果客户传入一个空的tweet呢？如果用户不再是一个有效的Twitter用户呢？

我们就这些关于加入tweets的考量做下面几个独立的测试。

```
TEST(ARetweetCollection, IgnoresDuplicateTweetAdded)
TEST(ARetweetCollection, UsesOriginalTweetTextWhenEmptyTweetAdded)
TEST(ARetweetCollection, ThrowsExceptionWhenUserNotValidForAddedTweet)
```

这样，你就能全面地查看测试名称，并且了解系统支持的确定行为。

3.6.3　使用测试来描述行为

你可以把测试想成一个示例，用它来描述或文档化系统中的行为。你可以通过以下两个方面来充分理解一个编写良好的测试：第一，测试名称，它概括了在特定上下文中系统表现出的行为；第二，测试语句本身，它精炼地展现了一个测试的行为。

c3/11/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, IgnoresDuplicateTweetAdded) {
   Tweet tweet("msg", "@user");
   Tweet duplicate(tweet);
   collection.add(tweet);

   collection.add(duplicate);

   ASSERT_THAT(collection.size(), Eq(1u));
}
```

测试名称提供了高层次的概括：应该忽略重复添加的tweet。但什么才算重复的tweet？忽略重复的tweet意味着什么？这个测试提示了一个简单的示例，清楚地回答了这两个问题。当一个tweet和另一个tweet完全相同时，算作重复；当加入一个重复的tweet后，tweet集合的大小不变。

越是重视TDD的文档功能，越懂得高质量测试的重要性。测试的文档功能是TDD的附属产物。为了确保在单元测试中的投入得到良好的回报，必须保证其他人能够很容易地理解测试，否则你的测试就是在浪费他们的时间。

良好的测试可以通过全面地记录系统行为来节省时间。只要所有的测试都通过，它们就准确地传达了系统的内部行为。文档也不会过时。

3.6.4　保持简单

无谓的复杂性带来的成本是无止境的。相信你曾花费无数个小时去解读一个复杂的成员函数或错综复杂的设计。大多数时候，你都可以编写一个更简单的解决方案，从而节省每个人的时间。

有很多原因会导致开发人员制造出无谓的复杂性。

- **时间压力**。“我们只需要发布代码，然后继续开发。我们没有时间把事情做得更好。”没有时间是迟早的事，因为你将耗费十倍的时间做任何事情。轻率使复杂性不断积累，这将在多个方面（代码可理解性、修改代码的开销、构建时间）让你速度变慢。
- **缺乏学习**。在追求更好的代码的同时，也要勇于承认制造出来的不良代码。这意味着你必须理解两者的差别。你可以通过团队成员结对和审阅的方式获得真实的反馈。学习怎样识别设计缺陷和代码坏味。学习怎样以更好的方式去纠正它们，可以找几本关于创建简洁设计和代码的书来学习。
- **已有的复杂性**。一个设计繁杂的已有代码库会让你在添加新行为时束手束脚。过长的方法会催生更冗长的方法，高耦合的设计也是孕育更高耦合设计的温床。
- **害怕改代码**。如果未经测试的话，你不会一直写出正确的代码。“如果没有出现问题，就不要去修复。”害怕修改代码会阻碍向更好的、可持续的设计进行重构。使用TDD的话，每个通过的测试都会提供改善代码的机会（或者至少不会让代码退化）。第6章将会专门讨论如何编写正确的代码。
- **臆测**。“客户和账号之间可能存在多对多的关系，所以现在就把这个添加到系统中。”或许大多数时候你是对的，但是你要承受过早引入的复杂性。有时候，你会选择另外一条思路，这时，你将不断地为这个（无用的）额外的复杂性买单，或者至少也要花费大量精力移除这个*没必要的复杂性*。相反，你应该在真正需要的时候再考虑上述做法。通常，这样做不会带来额外的开销。

保持简单是你在持续变化的环境中的生存之道。在敏捷开发里，每次迭代都会交付新特性，其中一些你可能从未考虑过。这是个挑战；如果系统不能适应新的改动，你就需要把新功能强加进系统。为此，最好的防范办法就是保持简单的设计：代码易读、没有冗余、没有无谓的复杂性。具有这些特征的系统会最大程度地降低维护成本。

3.6.5　恪守测试驱动开发周期

如果你未能遵循“红–绿–重构”周期，将会为此付出代价。参见3.5节来了解为什么第一时间注意到测试失败很重要。很明显，如果你期望通过的测试失败了，这意味着添加的代码不能正常工作。更重要的是，如果不经历重构阶段的话，你的设计质量会降低。不能恪守TDD规范流程会让你的开发速度变慢。

3.7　成功运用测试驱动开发的方法

前一节中重点讲述了成功运用TDD的哲学。这一节中将讨论各种具体技巧，这些技巧将帮助你在TDD的道路上顺利前行。

3.7.1　下一个测试是什么

既然已经开始学习TDD，那么萦绕在你心头上的最大问题之一就是：下一个测试应该怎么写？本书中的示例就这个问题给出了一些答案。

其中一个答案是：如果一个测试需要的产品代码最少，那么就写这个测试。但这到底是什么意思呢？

Bob大叔设计了一个体系，用于将每步变换分类（参考10.4节）。所有的变换按从最简单（最高优先级）到最复杂（最低优先级）安排优先级顺序。你要做的就是选择一个最高优先级的变换，然后为此变换写一个测试。按照变换优先级增量地进行，就能得到一个理想的测试顺序。上述内容就是前提：*变换优先级假设*（Transformation Priority Premise，TPP）。

如果你觉得TPP听起来很复杂，那是因为它本身就复杂。它是一个理论。目前而言，它证明了许多面向算法的解决方案是有意义的。

除此之外，你还可以通过回答下面的问题作出决定。

- 从逻辑上来说，哪个行为最具意义？
- 你可以验证这个有意义的行为中的哪个最小集合？
- 你能写一个测试来说明当前的行为不完备吗？

让我们测试驱动开发一个名为SQL的类，这个类的工作就是根据数据库表的元数据生成一些SQL语句（`select`、`insert`、`delete`，等等）。

我们可以通过每个新的测试获得有意义的行为。也就是说我们不会直接测试驱动开发getter、setter或构造函数。（在开发有用行为时会按需地编写它们。）生成操作数据表的SQL语句的操作似乎有用且足够小。我们先从`drop table`或`truncate table`开始写起。

测试名称	GenerateDropUsingTableName
实现	return “drop” + tableName_

这很简单，只需要几分钟就能编好。我们能很快地编写好truncate。

目前，生成SQL语句好像只是把数据表名称追加到一个命令字符串后，这提示我们可以重构，以简化drop和truncate语句的创建。（示例中，变量Drop和Truncate是常量字符串，每个尾部都有一个空格。）

3

```
std::string dropStatement() const {
   return createCommand(Drop);

}

std::string truncateStatement() const {
   return createCommand(Truncate);
}

std::string createCommand(const std::string& name) const {
   return name + tableName_;
}
```

我们也注意到了目前我们做得还不够。如果客户代码传入的数据表名称为空的话，代码处理起来就会出问题。

有时在处理所有的常规路径前，提前考虑异常情况是十分必要的。有时候，你也可以稍晚一些再来处理这些情况。通常情况下，你在很短的时间内就可以编写出针对异常情况的测试，仅仅需要少量的非侵入式的代码改动，就可通过这个测试。

测试名称	ConstructionThrowsWhenTableNameEmpty
实现	if (tableName_.empty()) throw …

我们选择select语句作为下一步有意义的增量代码。最简单的情形就是支持select*。

测试名称	GenerateSelectStar
实现	return createCommand(SelectStarFrom)

支持列查询很重要，因为大多数开发者认为相比于select*，列查询是更好的实践。

测试名称	GenerateSelectWithColumnList
实现	return Select + columnList() + From + tableName_

现在遇到了稍微复杂的情况。可能得花上几分钟来实现columnList()，但仍然不需要大动干戈，测试就能通过。

这里的select语句并不完备。我们进一步扩展使其支持where条件语句。

测试名称	GeneratesSelectWhereColumnEqual
实现	return selectStatement() + whereEq(columnName, value)

或许可以把GeneratesSelectWithColumnList的实现放到一个名为`std::string selectStatement() const`的成员函数里。接下来的测试GeneratesSelectWhereColumnEqual可以很容易地复用`selectStatement()`。这也是我们想要的结果：每一个测试都依赖于之前的基础构建，这样产生的影响最小。

久而久之，对于通过一个给定的测试，你的思路就会变得明朗起来。你的工作就是让自己的生活轻松一些，所以最好选择实现起来需要最少的增量代码的测试。

偶尔，你在选择下一个测试时也会作出不理想的选择。参见3.5.5节的示例。乐于回滚并丢弃掉少量的代码，能帮助你更好地学习到怎样在实现的过程中探寻更多的增量路径。

3.7.2　十分钟限制

TDD依赖较短的反馈周期。你只要恪守“红–绿–重构”周期，就能在TDD中做得很好，但依然有可能陷入困境。偶尔，你会很难让一个测试通过，或者在尝试清理代码时破坏一些测试，又或者需要借用调试器来探明到底发生了什么。

挣扎在所难免，但是可以限定遭受痛苦的时间。从上一个测试通过算起，不要超过十分钟。有一些开发者甚至用上了定时器。当然，不需要对时间限定得如此精确，但是当你尝试的解决方案有失偏颇时，做到这一点很重要。

如果限定的时间到了，那么丢掉你之前的结果然后重新来过。好的版本控制工具（如Git）让回滚变得容易、迅速、有效，而且很安全。（如果你不使用Git，那么可以考虑在本地使用一个桥接工具，如git-svn。）

不要过分纠结代码，特别是和现在做的不相干的代码。大可忽视它们。不要写超过十分钟的代码，否则，你的方案很可能质量不高。休息一下，放空思绪，然后重新以一个全新的视角来看问题。

如果你被之前无法工作的东西阻碍了，那么这一次可以采取更小的步伐来看看到底是哪里出了问题。你也可以加入额外的断言来验证有待商榷的预想。这样至少可以找出导致问题的那行代码，也有可能构建出更好的解决方案。

3.7.3　代码缺陷

代码总会有缺陷的。这无法避免。但是，TDD有可能让你的新代码接近零缺陷。我曾经见过一个团队，在产品发布的前十一个月里，其缺陷报告中只有十五个代码缺陷。其他使用TDD取得成功的案例比比皆是。

使用TDD几乎不会出现一些愚蠢的逻辑错误。那还会有哪些呢？其他的事情也会出现问题：没人预料到的条件、不同步的外部事情（例如，配置文件），以及多个方法或类合在一起时表现出的奇怪行为。如果规范有问题也会出错，包括因粗心而忽略掉一些东西，你和客户间的误解。

TDD不是银弹，却有助于消除一些人人都会犯的逻辑错误。（更重要的是，它是精细打磨系统设计的绝妙方法。）

当QA或客户支持团队当面给你指出一个代码缺陷时，你该怎么做？嗯，测试驱动！你或许可以先写一些简单的测试来探测系统行为。专注于相关代码的测试能帮助你更好地理解代码是怎样工作的，也有助于解读代码缺陷。有些时候，你可以保留这些测试，并把它们用作特征测试[①]（参见第8章）。有时候则可以丢弃这些类似一次性工具的测试。

一旦查明问题根源，也不要简单修复，继续做其他事情吧。我们是在测试驱动开发！相反，写一个你认为能模拟暴露缺陷的行为的测试。确保测试失败（红），修复它（绿），重构。

3.7.4 禁用测试

正常情况下，在测试驱动开发时一次只关注一件事情。个别情况下，当你忙于让第一个测试通过时，第二个测试失败了。如果另一个测试失败，这意味着你的第一个测试没能遵守“红–绿–重构”周期（除非这两个测试是因为同一个原因而失败）。

为了不让第二个失败的测试分散你的注意力，可以让它暂时失效。注释掉测试代码是可以的，但更好的方法是把测试标记为禁用。许多工具有显式地禁用测试的功能，并且在运行所有测试时提示你哪些是被禁用的。这个提醒功能可以避免误将测试禁用提交。

在Google Mock中，你可以给测试名称加上`DISABLED_`前缀来禁用测试。

```
TEST(ATweet, DISABLED_RequiresUserNameToStartWithAnAtSign)
```

当运行测试集时，Google Mock会在末尾打印出提示，这样就可以知道哪些测试被禁用了。

```
[----------] Global test environment tear-down
[==========] 17 tests from 3 test cases ran. (2 ms total)
[  PASSED  ] 17 tests.

   YOU HAVE 1 DISABLED TEST
```

不要提交被禁用（或注释掉）的测试的代码，除非你有充足的理由。集成后的代码应该反映系统的当前功能。注释掉的测试（或产品代码）会浪费其他开发者的时间。“测试被注释掉是因为没有这个行为了吗？这个测试有问题？处于变动中？这个测试运行太慢以至于需要的时候才启用它？我是不是应该找个人讨论一下这个测试？”

---

① 特征测试是用来刻画遗留软件系统中特定行为的测试。更多细节请参考https://en.wikipedia.org/wiki/Characterization_test。——译者注

3.8　结束语

在本章中我们学习了TDD的重要基础知识，包括它和单元测试的不同点，怎样遵循TDD周期（以及当事情超出预期时应该怎么应对），成功运用TDD的思维和方法。本章讲到的TDD背后的理念及哲学为继续学习TDD打下了基础。在下一章中我们将学习怎样运用这些理念来实现实际的测试。

第 4 章 测试结构

4.1 开场白

到目前为止，你应该对TDD的流程和概念有了深刻的理解。本章将深入讨论实现测试的具体细节，包括：文件组织、fixture、setup、teardown、过滤器、断言和基于异常的断言，以及其他一些零散的知识。

4.2 组织方式

从文件和逻辑方面着眼，应该怎样组织测试呢？在本节中，你将学到怎样用fixture组织测试，以及怎样利用setup和teardown这样的钩子函数。你也会学到如何使用Given-When-Then（又称Arrange-Act-Assert）的概念来组织测试内部。

4.2.1 文件组织

在测试驱动开发相关行为时，我们会将相关的测试定义在同一个测试文件中。例如，为了测试驱动开发一个RetweetCollection类（在RetweetCollection.cpp/.h中实现），从RetweetCollectionTest.cpp开始。不要立刻给测试创建一个头文件，这是吃力不讨好的。

最终你可能需要多个测试文件来验证相关的行为，也可能需要用一个测试文件覆盖多个地方的行为。不要拘泥于一个类一个测试文件这样的形式。4.2.3节将讲述一个类有多个测试文件的原因。

基于所包含的测试来给文件命名。概括相关的行为，并据此给测试命名。选定一个命名体系，诸如BehaviorDescriptionTest.cpp、BehaviorDescriptionTests.cpp或者TestBehavriorDescription.cpp。只要你的代码一致地遵守此标准即可。一致的命名约定可以让开发人员更容易地找到需要的测试。

4.2.2　fixture

大多数单元测试工具都支持将逻辑上相关的测试分组。在Google Mock中，你可以使用Google所谓的测试用例名称来将相关测试分组。下面的测试所属的测试用例名为ARetweetCollection。IncrementsSizeWhenTweetAdded是此测试用例中的一个测试。

```
TEST(ARetweetCollection, IncrementsSizeWhenTweetAdded)
```

相关的测试运行时需要相同的环境。你会发现许多测试都需要公共的初始化或辅助函数。许多测试工具能够让你定义一个fixture——一个跨测试可重用的类。

在Google Mock中，你可以定义一个派生自::testing::Test的fixture。通常是在测试文件的开始定义fixture。

```
using namespace ::testing;

class ARetweetCollection: public Test {
};
```

下面的两个测试有些重复代码，它们都创建了RetweetCollection的一个实例。

```
TEST(ARetweetCollection, IsEmptyWhenCreated) {
➤   RetweetCollection collection;

    ASSERT_THAT(collection.isEmpty(), Eq(true));
}

TEST(ARetweetCollection, IsNoLongerEmptyAfterTweetAdded) {
➤   RetweetCollection collection;
    collection.add(Tweet());

    ASSERT_THAT(collection.isEmpty(), Eq(false));
}
```

你可以在fixture里定义一次RetweetCollection，进而去掉RetweetCollection的局部定义。

```
class ARetweetCollection: public Test {
public:
➤   RetweetCollection collection;
};
```

如果想让测试能够访问fixture类的成员变量，那么就需要将TEST宏替换为TEST_F（尾部的F代表fixture）。下面是整理过的测试：

```
TEST_F(ARetweetCollection, IsEmptyWhenCreated) {
    ASSERT_THAT(collection.isEmpty(), Eq(true));
}

TEST_F(ARetweetCollection, IsNoLongerEmptyAfterTweetAdded) {
    collection.add(Tweet());

    ASSERT_THAT(collection.isEmpty(), Eq(false));
}
```

测试用例的名称必须与fixture的名称一样！如果没有使用带_F的宏，编译就会出错，指明测试引用了定义在fixture中的成员变量（这个例子中就是变量collection）。

阅读测试的人通常不需要知道创建collection变量这个细节就能理解测试。在这个例子中，将其移进fixutre可以避免分散注意力。如果你也这样将一些代码从测试中移入fixture，那么建议重新审阅一下测试。如果其意图依然清晰，那很好。如果不是，回滚你的改动，或者修改变量名称让其意义变得明确。

你可以而且应该将相同的函数移至fixture，特别是在没有将测试纳入到一个命名空间的情况下。

4.2.3 Setup 与 Teardown

如果测试用例中的所有测试需要一条或更多的相同初始化语句，那么可以将它们写在fixture类的初始化函数中。在Google Mock中，必须将此函数命名为SetUp（它覆写了基类::testing::Test中的虚函数）。

对于属于同一个fixture的测试，Google Mock都会创建一个新的、独立的fixture类实例。这种隔离有助于减少测试在执行过程中相互干扰而导致的问题。这也暗示着每个测试必须从头创建自己的上下文，并且这些上下文在测试之间相互独立。创建完fixture类实例后，Google Mock会执行SetUp()中的代码，然后执行测试。

下面两个测试向集合中加入一个Tweet对象用以创建一个初始上下文：

c3/14/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, IsNoLongerEmptyAfterTweetAdded) {
   collection.add(Tweet());
   ASSERT_FALSE(collection.isEmpty());
}

TEST_F(ARetweetCollection, HasSizeOfOneAfterTweetAdded) {
   collection.add(Tweet());
   ASSERT_THAT(collection.size(), Eq(1u));
}
```

我们为RetweetCollection测试定义一个新的fixture来代表有一个tweet的集合。我们使用ARetweetCollectionWithOneTweet作为fixture的名称。

c3/15/RetweetCollectionTest.cpp

```
class ARetweetCollectionWithOneTweet: public Test {
public:
   RetweetCollection collection;
   void SetUp() override {
      collection.add(Tweet());
   }
};
```

用公共的初始化代码创建出的代码非常易读。

c3/15/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollectionWithOneTweet, IsNoLongerEmpty) {
   ASSERT_FALSE(collection.isEmpty());
}

TEST_F(ARetweetCollectionWithOneTweet, HasSizeOfOne) {
   ASSERT_THAT(collection.size(), Eq(1u));
}
```

注意，在移除重复代码时不要移除一些对理解测试至关重要的信息。fixture的名称清晰地描述出上下文了吗？ARetweetCollectionWithOneTweet的意思似乎不言自明，但话说回来，自己说服自己是比较容易的。你可以问问其他程序员。如果非得阅读fixture的初始化代码才能理解它的意图，那么最好找个方法让测试更加明了。

第一个测试最初的名称是*用例名和测试名*的组合：ARetweetCollection.IsNotLongerEmpty-AfterTweetAdded（Google Mock称此组合为测试的*全名*）。现在，测试的全名为ARetweetCollection-WithOneTweet.IsNoLongerEmpty。命名fixture让其能描述上下文后，我们就不必将冗余的描述信息放到每个测试名称中。

现在还有另外一个可以整理的测试。

c3/15/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, IgnoresDuplicateTweetAdded) {
   Tweet tweet("msg", "@user");
   Tweet duplicate(tweet);
   collection.add(tweet);
   collection.add(duplicate);

   ASSERT_THAT(collection.size(), Eq(1u));
}
```

第一个加入到测试里的Tweet对象和其他测试里的Tweet对象不同。它为Tweet的构造函数的两个参数都赋值了，而其他的Tweet对象是以默认参数构造的。但是我们并不真正关心测试中集合里的Tweet对象是什么，只要集合中包含一个Tweet就可以了。我们为所有的测试选择更有趣的Tweet。

c3/16/RetweetCollectionTest.cpp

```
class ARetweetCollectionWithOneTweet: public Test {
public:
   RetweetCollection collection;
   void SetUp() override {
➤     collection.add(Tweet("msg", "@user"));
   }
};
```

但是在IgnoresDuplicateTweetAdded测试中创建第二个Tweet对象时，需要引用这个tweet。我们想在ARetweetCollectionWithOneTweet中加入一个成员变量。这意味着必须将之声明为指针类型。我们可以使用普通的C++指针，并在teardown函数中删除它。

c3/17/RetweetCollectionTest.cpp

```
class ARetweetCollectionWithOneTweet: public Test {
public:
   RetweetCollection collection;
   Tweet* tweet;
   void SetUp() override {
      tweet = new Tweet("msg", "@user");
      collection.add(*tweet);
   }

   void TearDown() override {
      delete tweet;
      tweet = nullptr;
   }
};
```

teardown函数本质上是setup函数的逆过程。每个测试后它都会执行一次，即便当测试抛出异常时也不列外。你可以将teardown函数用于清理工作：释放内存（就像这个示例中一样）、释放开销大的资源（如数据库连接），或者清理一些其他的状态，例如存储在静态变量中的数据。

如果使用指针，那么就需要稍微改动一下测试，因为现在`tweet`变量是指针类型。这样做的好处是可以将测试的代码从五行缩减至三行，而依然可以反映我们需要测试的东西。

c3/17/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollectionWithOneTweet, IgnoresDuplicateTweetAdded) {
➤  Tweet duplicate(*tweet);
   collection.add(duplicate);

   ASSERT_THAT(collection.size(), Eq(1u));
}
```

下面的fixture版本使用了智能指针：

c3/18/RetweetCollectionTest.cpp

```
class ARetweetCollectionWithOneTweet: public Test {
public:
   RetweetCollection collection;
➤  shared_ptr<Tweet> tweet;

   void SetUp() override {
➤     tweet = shared_ptr<Tweet>(new Tweet("msg", "@user"));
      collection.add(*tweet);
   }
};
```

初始化的代码适用于所有相关的测试。如果只用少数几个测试来设置上下文反而容易造成不

必要的困惑。当一些测试需要一个tweet，而其他测试不需要时，最好再创建一个fixture，并且把测试合理地分开。

在创建额外的fixture时，不要犹豫。但是每创建一个fixture，判断一下是不是需要显现出产品代码中的设计缺陷。如果需要两个不同的fixture的话，这有可能意味着你正在测试的类违反了单一责任原则，你可能需要将它们拆分为两个类。

4.2.4　Arrange-Act-Assert/Given-When-Then

测试都有相同的流程。首先需要设置好合适的条件，然后执行代表要验证的行为的代码，最后验证结果是否和期望的一样。（有些测试可能需要一些清理工作。例如，一个测试可能需要关闭之前打开的数据库连接。）

测试应当尽可能地直接反映其测试意图。这就意味着阅读测试的人不需要细细品读测试中的每一行，就能很容易地理解测试的基本构成：测试的初始化（Arrange）、测试的行为（Act），以及怎样验证行为结果（Assert）。

Arrange, Act, Assert（AAA，通常读作triple-A）助记词是由Bill Wake[①]发明的，它提醒你直观地去组织测试以便能够快速阅读。看看下面的测试，你能快速看出哪些代码行是用于设置上下文的，哪些是用来执行所要验证的行为的，哪些是和真正的断言相关的吗？

c3/14/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, IgnoresDuplicateTweetAdded) {
   Tweet tweet("msg", "@user");
   Tweet duplicate(tweet);
   collection.add(tweet);
   collection.add(duplicate);
   ASSERT_THAT(collection.size(), Eq(1u));
}
```

如果代码行攒簇在一起，那么就需要多花些时间来看清楚它们到底在测试什么。现在来对比一下将Arrange、Act、Assert区分开的版本。

c3/13/RetweetCollectionTest.cpp

```
TEST_F(ARetweetCollection, IgnoresDuplicateTweetAdded) {
   Tweet tweet("msg", "@user");
   Tweet duplicate(tweet);
   collection.add(tweet);

   collection.add(duplicate);

   ASSERT_THAT(collection.size(), Eq(1u));
}
```

① http://xp123.com/articles/3a-arrange-act-assert

一些测试驱动开发者喜欢助记词Given-When-Then。给定（Given）一个上下文，当（When）测试调用一些行为，然后（Then）验证结果。（清理工作不能作为助记词的一部分，因为此工作对理解测试验证的行为意义不大。）你也可能听过Setup-Execute-Verify（-Teardown）这样的表述。Given-When-Then表述法稍微侧重强调验证行为，而非测试执行。这也和验收测试驱动开发（Acceptance Test-Driven Development，ATDD）中的概念吻合（参见10.3节）。

AAA并不是让人脑洞大开的理念，但是如果一直使用它，会让阅读测试的人轻松一些。

4.3 快速测试、慢速测试、过滤器和测试集

如果你编写的是小而独立的代码单元，那么每个测试都会运行得快如闪电。通常一个测试在一台配置完备的电脑上的运行时间不到一毫秒。以这种速度，几分钟内至少可以运行几千个测试。

如果与一些外部慢速资源（如数据库或其他慢速服务）交互的话，那么测试就会变慢。单单建立数据库连接就可能花费50毫秒。如果大部分测试必须与数据库交互，那么需要几分钟才能运行完几千个测试。有些工作室需要半个多小时才能运行完所有测试。

我们将在第5章中学习怎样打破对慢速组件的依赖。为什么构建快速的测试可以决定能否成功运用TDD呢？

TDD的核心目标就是尽可能频繁地获得较多的反馈。当你修改了一点代码时，会想马上知道改动是否正确。你是否破坏了某处的代码呢？

你每做出一次小的改动，都需要运行所有的单元测试。TDD最大的好处是它能让你在短时间内获得有用的反馈。如果构建的测试运行得快，那么如前所述，完全有可能在几秒内运行完所有的测试。如果等待的时间足够短，那么经常运行所有的测试也是合理的。

如果运行完所有的测试需要的时间不止数秒，那么就不要频繁地运行它们。如果测试需要运行两分钟，你会多久运行一次呢？或许1小时5次。如果需要20分钟呢？可能一天运行的次数掰手指都能数过来了吧！

一旦反馈周期变长，TDD的威力就会减弱。获取反馈的间隔时间越长，那么你写的代码出问题的可能性就越大。通常在编写代码时很容易引入或大或小的问题。相比之下，每做一个小改动就运行一次测试，就能逐个解决这些问题。在写了几分钟的代码后，如果测试发现了问题，你能很容易地定位到导致该问题的代码。如果你编写了几行晦涩难懂的代码，那么可以容易且安全地整理它们。

运行慢的测试对TDD来说是个问题，所以一些人不再把它们称为单元测试，而是称其为集成测试（参见10.3节）。

运行测试的一个子集

你可能已经有一个测试集（Test suite）来验证系统的一部分行为。这些测试很有可能运行得不是很快，因为大部分已有的系统都依赖于许多慢速协作者。

我们对于运行速度较慢的测试集的第一反应是少运行几次。如果你想要做测试驱动开发，那么这个策略是行不通的。我们的第二反应是只运行其中的一个测试子集。虽然这个方法不是那么理想，但也可行，前提是了解其背后的意义。

Google Mock可以通过指定一个测试过滤器轻松地做到只运行一个测试子集。你可以把测试过滤器作为执行测试的命令行参数。过滤器的语法是：测试用例名.测试名称。例如，如果想要运行一个特定的测试，可以使用下面的命令：

```
./test --gtest_filter=ATweet.CanBeCopyConstructed # weak
```

但是不要养成每次只运行一个测试的习惯。可以使用通配符（*）运行多个测试。让我们运行一下ATweet测试用例中所有的测试。

```
./test --gtest_filter=ATweet.* # slightly less weak
```

如果你在开发和tweet相关的类，那么最好找到一个方法来运行所有和tweet相关的测试。使用多个通配符吧！

```
./test --gtest_filter=*weet*.*
```

这个过滤器包含了RetweetCollection中所有的测试。（我省略写T，因为它不能和Retweet-Collection中的小写t匹配。）

如果你想在不运行Tweet类的任何构造测试的情况下运行和tweet相关的所有测试，该怎么做呢？这时，Google Mock允许你创建复杂的过滤器。

```
./test --gtest_filter=*Retweet*.*:ATweet.*:-ATweet*.*Construct*
```

可以使用冒号（:）来分隔不同的过滤器。如果Google Mock遇到一个负号（-），那么其后的所有过滤器匹配的测试都将被忽略。在上面的例子中，-ATweet*.*Construct*告诉Google Mock忽略名称中含Construct的所有ATweet测试。

只运行一个测试子集会出现什么问题呢？你在短时间内运行的测试越多，就越有可能知道什么时候引入了缺陷。运行的测试越少，那么找到一个缺陷的平均时间就越长。通常，引入一个缺陷到发现它的时间间隔越长，修复它所需的时间就越久。原因很简单。第一，如果这段期间你做过其他事情，那么通常需要花费额外的时间来理解原先的解决方案。第二，新增加的代码会增加理解难度和修复缺陷的难度。

许多单元测试工具（Google Mock除外）直接支持永久地指定任意测试集。例如，CppUnit提供了一个TestSuite类，它允许你以编程的方式向一个测试集合中加入测试来运行。

测试分类

最近，我在为一个客户工作，他们有一个相当强大的C++系统。在我加入团队之前，他们刚刚开始使用TDD，并且已经开发了几百个测试。但是大部分测试运行缓慢，运行完所有测试需要大约3分钟，对于这个测试数目来说，这需要的时间过长。大部分测试会花费300毫秒或更长时间。

我们在Google Mock的基础上快速地实现了一个测试监听程序，它的工作就是生成一个名为slowTest.txt的文件，其中包含运行超过10毫秒的测试名称列表。然后，我们修改Google Mock，使其可以从一个文件中读入一些过滤器。这个改动本质上为Google Mock提供了测试集支持。此外，我们还修改了Google Mock来支持运行一个过滤器的**相反**测试。这样一来，既可以使用slowTests.txt运行**慢的**测试，也可以运行**快的**测试。结果是慢速测试集花费了三分钟的大部分时间（参见10.3节），而快速测试集只花费了几秒钟。

进一步修改Google Mock，使运行时间超过给定值的测试失败（通过一个名为`slow_test_threshold`的命令行选项）。我的客户把这些都配置到持续集成系统中，先运行**快速**测试集，运行得太慢就会失败，之后再运行慢速测试集。

我们建议程序员在开发过程中不断地运行这个快速测试集。开发人员在运行快速测试时会指定一个`slow_test_threshold`，以便在一个慢速测试加入时收到通知。一旦要提交代码，就需要运行慢速和快速测试集。我们让开发者想办法消除使测试运行缓慢的不良依赖，以此不断地减小slowTests.txt的大小（参见第5章）。

这个团队理解了怎么去做，并且最终大部分测试都能快速运行。上次我听说，他们已经成功地快速开发系统了。

我已经通过类似测试分类的流程帮助了一些团队。通常使用的是80-20原则：20%的测试花费80%的运行时间。将快速测试和慢速测试分开能够快速提高团队的TDD能力。

定义测试集的能力为分割测试提供了基础，这样就可以只运行那些速度快的测试。发现慢速的测试，就可以逐渐地把慢速测试转换为快速测试。

通常，你是需要集成测试（那些必须与外部服务集成的测试，参见10.3节）的。这些测试运行得比较慢。一旦你拥有自定义任意测试集的能力，就可以维护多个测试集，用在持续集成过程中。

4.4　断言

断言可以将一个普通的测试变成自动化的测试。如果没有断言，那么Google Mock只是执行一段代码而已。如果想要验证一段代码是否能正确工作，则需要人工查看结果（可以通过单步调试或打印出变量内容）。

人工验证测试结果是耗时的。TDD能生成自我验证的（参见7.2节）自动化的单元测试，这

将省去查看结果的工作。人工查看是有风险的，且单调乏味。没有比人工为应用程序编写GUI更费时间的事了，因为你需要一次又一次地去验证本可以通过单元测试自动验证的东西。

当测试框架运行单个测试时，它会从头到尾执行测试代码段中的语句。每遇到一个断言，都意味着要去验证一些期待的结果。如果断言的条件不满足，那么测试框架就中止测试。（通常，断言中会抛出测试框架可以捕获的异常。）测试框架会保存测试失败的信息，运行teardown逻辑，然后接着运行下一个测试。

有些工具，包括Google Mock，提供了另一种断言机制，它允许测试在遇到断言失败的情况下继续运行。这些断言又称作*非致命性断言*，与*致命性断言*相对，后者会中止测试。

在产品测试中应该使用致命性断言。一旦断言失败就应该中止测试。当断言验证的假设失败时，继续执行测试的意义不大。如果追求一个测试一个断言（参见7.3节），那么在唯一的断言后就很少会再有代码了。

当你在设计测试或解读测试失败时，很有可能用到额外的断言来用作探查。成员变量初始化成特定的值了吗？配置设好了预先的状态了吗？如果非致命性断言失败了，测试还会继续运行，这或许额外给你提供了一些有用的信息。通常，在准备提交代码前会移除探查时用的断言。

4.4.1 经典的断言形式

大部分测试框架都支持我所谓的经典断言形式。它们沿袭了SUnit风格，SUnit在20世纪90年代构建，是Smalltalk的单元测试框架。使用这种形式并无不妥，但是你或许会考虑形式更加多样的断言形式，即Hamcrest断言（参见4.2.2节）。由于你会遇到大量使用经典断言形式的代码，因此学习这两类断言是十分必要的。

下表列出了Google Mock中的两个主要断言。其他框架也会使用类似的名称。

形　　式	描　　述	示　　例
ASSERT_TRUE(表达式)	表达式返回假（或0）时，测试失败	ASSERT_TRUE(4<7)
ASSERT_EQ(期待值，实际值)	期待值和实际值不等时，测试失败	ASSERT_EQ(4, 20/5)

Google Mock和大多数C++单元测试框架一样，通过宏来实现断言的行为。通常都要经过重载来实现相等的断言，用来支持所有的基本类型比较。

大部分测试框架会提供一些额外的断言形式来提升表达力。例如，Google Mock支持`ASSERT_FALSE`（当表达式返回假时，断言成立）和一些关系型断言，如`ASSERT_GT`（当第一个参数大于第二个参数时，断言成立）。可以用`ASSERT_LT(4, 7)`来替代上表中的第一个断言示例。

4.4.2 Hamcrest断言

单元测试工具提供了一个小且固定的经典断言集合。大部分常用的断言，如比较两个值相等

（Google Mock中的ASSERT_EQ），是符合习惯的用法。你一定要把期望值放在前面。如果你要大声朗读一个断言，大致是这样的：断定期望值5等于实际值x。这没什么的，因为你日后将阅读成千上万个这样的断言，但是这种表达会稍微影响测试的可读性。同时，对于初学者来说也容易误将实际值和期望值弄反。

几年前引入Hamcrest断言就是为了提升测试的表达力、创建复杂断言的灵活性，以及测试错误所提供的信息。Hamcrest使用匹配器（matcher，Hamcrest是Matchers的衍生词）比较实际结果。匹配器可以组合成复杂但易懂的比较表达式。你也可以自定义匹配器。

几个简单的示例胜过千言万语，至少可以省去不少篇幅。

```
string actual = string("al") + "pha";
ASSERT_THAT(actual, Eq("alpha"));
```

这个断言从左至右读作：断定实际值等于"alpha"。对于一个简单的相等比较而言，区区几个额外的字符就能够提升可阅读性。

起初，Hamcrest断言貌似过于炫技。但是匹配器的价值在于它能极大地提升测试的表达力。许多匹配器既能减少所需的代码量，同时也能提升测试的抽象层次。

```
ASSERT_THAT(actual, StartsWith("alx"));
```

Google Mock的文档列出了一些内置的匹配器[①]。

使用它们时需要在测试文件中加入using声明。

```
using namespace ::testing;
```

否则本意用来提升表达力的断言读起来会有些啰嗦卡顿。如下：

```
ASSERT_THAT(actual, ::testing::StartsWith("al"));
```

Hamcrest断言在提升失败信息的可读性方面意义更大。

```
Expected: starts with "alx"
  Actual: "alpha" (of type std::string)
```

匹配器的组合能力使你用一行断言就能表达本来需要多行断言才能做到的事情。

```
ASSERT_THAT(actual,
      AllOf(StartsWith("al"), EndsWith("ha"), Ne("aloha")));
```

上述例子中的AllOf表明只有当所有匹配器都成功时，整个断言才算通过。因而actual必须以"al"开头，以"ha"结尾，且不等于"aloha"。

对于布尔值的断言而言，大部分开发者都会避开Hamcrest断言而使用经典的断言形式。

```
ASSERT_TRUE(someBooleanExpression);
```

① http://code.google.com/p/googlemock/wiki/V1_6_CheatSheet#Matchers

```
ASSERT_FALSE(someBooleanExpression);
```

总之，如果Google Mock内置的匹配器不能满足需求，你也可以自定义匹配器①。

4.4.3 选择正确的断言

测试中断言的目标是清晰地描述其之前代码执行的结果是否满足期望值。通常，断言需要明确。如果你知道一个变量的值应为3，那么就使用相等的断言来准确地声明。

```
ASSERT_THAT(tweetsAdded, Eq(3));
```

即使弱化一点的比较有时候也更具表达力，但是大部分时候还是要避免使用。

```
ASSERT_THAT(tweetsAdded, Gt(0)); // 避免广泛的断言
```

大部分断言都应该使用相等的形式。从技术上讲，ASSERT_TRUE是非常普适的，但是当断言失败时，相等断言（无论是不是Hamcrest）都能传递更好的信息。例如，当下面的断言失败时……

```
unsigned int tweetsAdded(5);
ASSERT_TRUE(tweetsAdded == 3);
```

失败信息提供了少量的信息。

```
Value of: tweetsAdded == 3
  Actual: false
Expected: true
```

但是如果使用下面的形式……

```
ASSERT_THAT(tweetsAdded, Eq(3));
```

失败的信息会告诉你断言的期望值和实际值。

```
Value of: tweetsAdded
Expected: is equal to 3
  Actual: 5 (of type unsigned int)
```

错误信息中的序列解释了为什么一直要保持期望值和实际值的顺序正确。如果一不小心把顺序弄反，就会出现令人混淆的错误信息，而解读此信息会浪费额外的时间。如果使用ASSERT_THAT()，那么实际值在前面。如果使用ASSERT_EQ()，那么期望值在前面。

4.4.4 浮点数比较

浮点数是实数的二进制表示，但是这种表示是不准确的。所以，两个浮点数计算出的结果有可能不完全相等，即便它们应当是相等的。下面是一个例子：

```
double x{4.0};
```

① http://code.google.com/p/googlemock/wiki/V1_6_CheatSheet#Defining_Matchers

```
double y{0.56};
ASSERT_THAT(x + y, Eq(4.56));
```

在我的机器上会收到下面的错误信息：

```
Value of: x + y
Expected: is equal to 4.56
  Actual: 4.56 (of type double)
```

Google Mock和其他工具提供了特殊的断言形式，可以用它们来比较两个浮点数，这种比较允许有一定的误差。如果两个浮点数的差大于这个误差，那么断言失败。Google Mock提供了基于ULP（Units in Last Place）的简单的比较方法。

```
ASSERT_THAT(x + y, DoubleEq(4.56));
```

如果够大胆，那么你可以在即将发布的下一版Google Test中指定误差。ULP和浮点数比较相对来说是复杂的话题（http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm）。

4.4.5 基于异常的测试

优秀的工程师应当知晓在代码执行时会出现哪些错误，也应当知道什么时候抛出异常，什么时候需要引入try-catch块来保护应用程序。唯有对代码执行路径了如指掌，方能知道该在什么时候处理异常。

在TDD中，先从一个失败的测试开始，然后将对异常的顾虑化作代码编写入系统。所得结果就是可以用作文档之用的测试，可以将此测试提供给不太清楚代码执行路径的开发人员。当异常发生时，可以在测试中找到可能出错的地方和将会发生的事情。拥有这些知识的客户端开发者可以很自信地使用你提供的类。

试想你现在在测试一段代码，它在某些情况下会抛出异常。作为专业人员，你的工作就是用一个测试来文档化这种情形。

有些单元测试框架允许你声明应该抛出的异常，如果异常没有抛出，则测试失败。使用Google Mock可以编写如下代码：

c3/12/TweetTest.cpp

```
TEST(ATweet, RequiresUserToStartWithAtSign) {
   string invalidUser("notStartingWith@");
   ASSERT_ANY_THROW(Tweet tweet("msg", invalidUser));
}
```

如果ASSERT_ANY_TRHROW宏内的表达式不抛出异常，那么测试失败。运行一下测试就会得到下面的错误信息。

```
Expected: Tweet tweet("msg", invalidUser) throws an exception.
  Actual: it doesn't.
```

下面是让这个测试通过的相应代码。

c3/12/Tweet.h

```
Tweet(const std::string& message="",
      const std::string& user=Tweet::NULL_USER)
   : message_(message)

   , user_(user) {
   if (!isValid(user_)) throw InvalidUserException();
}

bool isValid(const std::string& user) const {
   return '@' == user[0];
}
```

（isValid()的实现足够让这个测试通过了。这个实现假设Tweet的构造函数接受一个空字符串作为user的实参。那么，还需要我们写什么测试呢？）

如果你知道将要抛出异常的类型，那么可以指定它。

c3/12/TweetTest.cpp

```
TEST(ATweet, RequiresUserToStartWithAnAtSign) {
   string invalidUser("notStartingWith@");
   ASSERT_THROW(Tweet tweet("msg", invalidUser), InvalidUserException);
}
```

下面的错误信息会告诉你当期待的异常类型和实际抛出的不一致时会是怎样的。

```
Expected: Tweet tweet("msg", invalidUser) throws an exception
          of type InvalidUserException.
  Actual: it throws a different type.
```

如果你的单元测试框架不支持确定异常抛出的单行断言，那么也可以在测试中使用下面的方法。

c3/12/TweetTest.cpp

```
TEST(ATweet, RequiresUserNameToStartWithAnAtSign) {
   string invalidUser("notStartingWith@");
   try {
      Tweet tweet("msg", invalidUser);
      FAIL();
   }
   catch (const InvalidUserException& expected) {}
}
```

我们可以用多种方法达成此目的，但我还是喜欢这里所讲的技巧。这是TDD社区所喜好的惯用法。如果必须要验证异常抛出后的条件，那么可以使用try-catch块。例如，想要验证异常对象的文本信息。

c3/13/TweetTest.cpp

```
TEST(ATweet, RequiresUserNameToStartWithAtSign) {
```

```
        string invalidUser("notStartingWith@");
        try {
           Tweet tweet("msg", invalidUser);
           FAIL();
        }
        catch (const InvalidUserException& expected) {
           ASSERT_STREQ("notStartingWith@", expected.what());
        }
     }
```

注意ASSERT_STREQ的用法。Google Mock提供了四种断言宏（ASSERT_STREQ、ASSERT_STRNE、ASSERT_STRCASEEQ和ASSERT_STRCASENE），它们用于支持C风格（以'\0'结尾）的字符串，即char*变量。

4.5 探查私有成员

TDD新手必然会有两个疑问：我可以针对私有成员数据编写测试吗？私有成员函数呢？这两个既相关又迥异的话题会影响你的设计抉择。

4.5.1 私有数据

Tell-Don't-Ask设计理念说，你应该告诉一个对象去做一些工作，然后让它自主完成任务。如果在这期间频繁地过问对象，那么就违反了Tell-Don't-Ask原则。如果一个系统包含大量的对象查询，它是纠缠不清且极度复杂的。试想一下下面的情形：客户C查询对象S以获取信息，然后做了本来对象S可以完成的工作，再查询一下对象S，如此重复。这使客户C和对象S之间紧密交互。因为对象S并没有做本该做的工作，所以除客户C之外的其他客户也可能会查询相同的信息，并使用重复的代码逻辑来处理对象S的返回信息。

通常，当一个对象被要求完成某项任务时，它会把具体工作交由合作对象来处理。相应地，可以用一个测试来验证这个交互：合作对象接受到信息了吗？这可以使用测试替身（参见第5章）。这就是所谓的*交互测试*。

然而，并不是所有的交互操作都需要检测合作对象。例如，当测试驱动开发一个简单的容器时，你会想要验证添加进其内的任意对象。可以简单地通过公共接口来查询并用断言来验证答案。验证对象属性的测试称作*状态测试*。

让客户端程序知道他们把什么添加进一个对象是合理的。加一个访问函数。（如果你担心恶意客户端程序通过这个访问函数搞破坏的话，那么就让函数返回一个对象的副本。）

极少数情况下需要保存一些中间计算的结果，大体上，这些结果是调用函数产生的。为那些非公用接口的成员提供访问通道是可以接受的。你也可以将测试声明为待测试类的友元，但不要这么做。你可以添加简要注释来说明你的意图。如下所示：

```
public:
   // 仅仅为了测试的目的而暴露数据；避免直接用于生产：
   unsigned int currentWeighting;
```

将数据暴露出来仅仅是为了测试，这对许多人来说是不易接受的，但更重要的是清楚你的代码能正常工作。

然而，过量的状态测试显露了设计坏味①。无论什么时候，如果暴露数据仅仅是为了断言，那么就需要考虑用验证行为的方式来替代。参考第5章获取更多信息。

4.5.2　私有行为

做测试驱动开发时，所有东西都来自于类的公共接口设计。为了添加详细的行为，你测试驱动开发这个行为，就好像你的测试就是此类的产品客户。随着细节越来越多和代码越来越复杂，你自然就会觉得需要重构来提取其他的方法。你会想是不是应该针对这些提取出来的方法直接写相应的测试。

例如，一个图书馆系统定义了一个`HoldingService`类，它提供了图书的借出和归还接口。`CheckIn()`方法为图书的归还提供了合理（但有点乱）的高层次策略实现。

c7/2/library/HoldingService.cpp

```cpp
void HoldingService::CheckIn(
      const string& barCode, date date, const string& branchId)
{
   Branch branch(branchId);
   mBranchService.Find(branch);

   Holding holding(barCode);
   FindByBarCode(holding);

   holding.CheckIn(date, branch);
   mCatalog.Update(holding);

   Patron patronWithBook = FindPatron(holding);

   patronWithBook.ReturnHolding(holding);

   if (IsLate(holding, date))
      ApplyFine(patronWithBook, holding);

   mPatronService.Update(patronWithBook);
}
```

挑战来自`ApplyFine()`函数。最初实现`CheckIn()`方法的程序员是从只考虑一种情况的简单实现开始的，但是在情况变得复杂时，提取出了单独的函数。如下所示：

---

① 和代码坏味类似，设计坏味指的是设计里存在的糟糕方案，需要改进。

c7/2/library/HoldingService.cpp

```
void HoldingService::ApplyFine(Patron& patronWithHolding, Holding& holding)
{
   int daysLate = CalculateDaysPastDue(holding);

   ClassificationService service;
   Book book = service.RetrieveDetails(holding.Classification());

   switch (book.Type()) {
      case Book::TYPE_BOOK:
         patronWithHolding.AddFine(Book::BOOK_DAILY_FINE * daysLate);
         break;
      case Book::TYPE_MOVIE:
         {
            int fine = 100 + Book::MOVIE_DAILY_FINE * daysLate;
            if (fine > 1000)
               fine = 1000;
            patronWithHolding.AddFine(fine);
         }
         break;
      case Book::TYPE_NEW_RELEASE:
         patronWithHolding.AddFine(Book::NEW_RELEASE_DAILY_FINE * daysLate);
         break;
   }
}
```


围绕`ApplyFine()`的每个测试必须在一个上下文中运行，这就需要客户先借出一本书，然后再归还这本书。那么通过直接枚举测试`ApplyFine()`的所有代码路径岂不是更有意义？

我们应该认为直接为`ApplyFine()`写测试是非常不好的，因为这违反了信息隐藏的原则。更重要的是，我们应该察觉到示例中的设计缺陷。`ApplyFind()`还违反了单一责任原则：HoldingService应该只为客户提供高层次的任务步骤。每一个任务的实现细节应该出现在其他地方。从另外一个角度看，`ApplyFine()`还犯了feature envy[①]：它应该存在于另一个类，或许是Patron中。

大多数时候，当你觉得需要测试私有行为时，可以尝试将代码移到另一个类或为之创建的一个新类。

但是，不能将`ApplyFine()`全部移到Patron类中，因为它做了多件事情：它调用了另一个函数来计算图书过期天数，判断给定图书的类型，并计算对顾客的罚金。前两项任务需要访问HoldingService的其他功能，所以需要暂时保留在HoldingService中。但是我们可以独立计算出罚金。如下所示：

c7/3/library/HoldingService.cpp

```
void HoldingService::ApplyFine(Patron& patronWithHolding, Holding& holding)
{
   unsigned int daysLate = CalculateDaysPastDue(holding);
```

① feature envy，译为“依恋情结”，是众多代码坏味中的一种，可以参考《重构》一书。

```
    ClassificationService service;
    Book book = service.RetrieveDetails(holding.Classification());
    patronWithHolding.ApplyFine(book, daysLate);
}
```

c7/3/library/Patron.cpp

```
void Patron::ApplyFine(Book& book, unsigned int daysLate)
{
    switch (book.Type()) {
        case Book::TYPE_BOOK:
            AddFine(Book::BOOK_DAILY_FINE * daysLate);
            break;

        case Book::TYPE_MOVIE:
            {
                int fine = 100 + Book::MOVIE_DAILY_FINE * daysLate;
                if (fine > 1000)
                    fine = 1000;
                AddFine(fine);
            }
            break;

        case Book::TYPE_NEW_RELEASE:
            AddFine(Book::NEW_RELEASE_DAILY_FINE * daysLate);
            break;
    }
}
```

现在我们再来看一下移到新地方后的函数，貌似还是有点问题。查询图书类型还是有点feature envy嫌疑。`switch`语句也是有代码坏味的。用多态来取代`switch`使得我们可以创建更直接且更有针对性的测试。但目前而言，我们已经把`ApplyFine()`放到了一个可以公开直接测试它的理想位置。

> **提问**：如果这样编写程序的话，我最后会不会得到数千个特定功能的类？
>
> **回答**：当然会多出一些类，但还不至于数以千记。每个类会非常小，这样也易于理解/测试/维护，编译也很快！（参见4.3节。）
>
> **提问**：我不提倡创建更多的类。
>
> **回答**：这时你就开始真正利用OO（Object-Oriented，面向对象）了。当你创建更多单一目标的类并且每个类包含一些单一责任的方法时，你会发现更多的重用契机。重用一个庞杂的类是不可能的。相反，遵循单一责任原则的类会让你看到减少代码量的可能性。

如果你面对的是遗留代码呢？假设你要先从一个有几十个成员函数的类下手，其中许多还是私有成员。这跟私有行为类似，可以先简单地弱化访问关系，并为一些私有函数添加相应测试。（再一次提醒，不要担心类的泛滥，这不会发生。）将函数移至更好的场所。你可以参见第5章来获得解决遗留系统的良方。

4.6　测试和测试驱动：参数化的测试及其他方法

尽管测试驱动开发中有测试两个字，但是它更多地与设计有关，而非测试。在TDD过程中，你会编写单元测试，但它们基本就是一些副产品。貌似差别不大，但TDD的真正目的是让你一直保持设计整洁，这样在引入新的行为或改变现有行为时，你会更从容自信，并且不会太费力。

从测试的角度看，你寻求的是创建具有高覆盖率的测试。所写的测试有五类：无（zero）、有（one）、多（many）、边界（boundary）和异常（exceptional）情形。而从测试驱动角度来看，你写测试的目的是为了保证代码能够符合预定规范。虽然测试和测试驱动都能够保证你有足够的信心去发布代码，但是一旦对所构建的东西有足够的信心，就可以停止TDD。与此相反，优秀的测试人员会竭尽所能地去覆盖上面所说的五类情形。

你可以在测试驱动开发时编写额外的事后测试。但通常而言，一旦你认为你有一个正确且干净的实现，并且这个实现能覆盖你要支持的情形，那么就可以立刻停止。换句话说，在你想不出可以写出不能通过的测试时，就可以停止。

现在以罗马数字转换器（参见附录B）为例，它可以将阿拉伯数字转换为对应的罗马数字。优秀的测试人员可能至少会测试几十个转换，以确保能覆盖各种数字和组合。相反，在测试驱动开发解决方案时，我在测试完十几个转换后就可以停下来。此时，我有足够的信心保证你已经开发出了正确的算法，剩下的工作仅仅是完成阿拉伯数字到罗马数字的转换表。（在附录中，我测试驱动开发了更多的断言，以达到提升信心和示范的目的。）

许多代码级的测试工具起初都是用来支持写测试而不是用来做TDD的。所以，许多工具提供了复杂的特性，以便让测试变得更简单。例如，一些工具允许你在测试间定义依赖关系。如果你有一个运行很慢的集成测试集（参见10.3节），那么这确实是个很好的优化特性，可以把测试按照一定的顺序组织起来以便加快测试运行。（维护这种紧密耦合在一起的测试的成本也将增加。）但是，在测试驱动开发时，你追求的是快速且独立的测试，所以不需要这种测试间的依赖关系，因为它会让事情变得复杂。

偶尔需要或使用这些偏测试的特性也没什么不妥之处。但是，使用前先问自己几个问题：这个特性是不是让我偏离了测试驱动开发？这个特性和TDD的目标吻合吗？

本节将大致描述几个有吸引力的测试工具的特性。如果很想使用它们（即便在我劝说你不要使用之后），那么可以参考手头的测试工具来了解特性的细节信息。

4.6.1　参数化测试

罗马数字转换器（参见附录B）必须转换1~3999的数字。如果可以简单地遍历一系列期望的输入和输出，并把它们灌进一个能够接受一个输入和一个输出作为参数的测试，这样也不错。一些测试工具（包括Google Mock）的参数化测试特性就可以做到这一点。

下面通过一个简单的名为Adder的类来说明：

c3/18/ParameterizedTest.cpp

```
class Adder {
public:
   static int sum(int a, int b) {
      return a + b;
   }
};
```

下面是一个常规情况下测试驱动过程产生的测试，它驱动了`sum()`的实现：

c3/18/ParameterizedTest.cpp

```
TEST(AnAdder, GeneratesASumFromTwoNumbers) {
   ASSERT_THAT(Adder::sum(1, 1), Eq(2));
}
```

但是上述测试只覆盖了一种情形！没错，我们要对自己的代码有信心，不需要创建一堆额外的测试。

对于更复杂的代码，或许有着覆盖一大堆情形的测试会让我们更有信心。对于Adder而言，我们可以先定义一个衍生自TestWithParam<T>的fixture，这里的T是参数类型。

c3/18/ParameterizedTest.cpp

```
class AnAdder: public TestWithParam<SumCase> {
};
```

这里的参数类型为SumCase，用来描述两个输入值和一个期望的和。

c3/18/ParameterizedTest.cpp

```
struct SumCase {
   int a, b, expected;
   SumCase(int anA, int aB, int anExpected)
      : a(anA), b(aB), expected(anExpected) {}
};
```

有了这些，就可以写参数化的测试了。我们使用`TEST_P`宏来声明测试，其中`P`表示参数化。

c3/18/ParameterizedTest.cpp

```
TEST_P(AnAdder, GeneratesLotsOfSumsFromTwoNumbers) {
   SumCase input = GetParam();
   ASSERT_THAT(Adder::sum(input.a, input.b), Eq(input.expected));
}
SumCase sums[] = {
   SumCase(1, 1, 2),
   SumCase(1, 2, 3),
   SumCase(2, 2, 4)
};
INSTANTIATE_TEST_CASE_P(BulkTest, AnAdder, ValuesIn(sums));
```

最后一行代码使用事先准备好的参数运行测试。INSTANTIATE_TEST_CASE_P的第二个参数是fixture的名称，第三个参数是准备好的测试值。（第一个参数BulkTest是Google Mock加在测试名称前的前缀。）ValuesIn()函数指示在每次运行测试时，其输入的测试值是来自名为sums的数组（GeneratesLotsOfSumsFromTwoNumbers）。测试中的第一行调用了GetParam()函数，它返回输入的测试值（一个SumCase对象）。

酷！但是我做了十多年的TDD，却很少使用参数化测试。如果你需要测试大量的简单数据，那么参数化测试可能是一个很好的选择。例如，某个人给你一个充满了数据的电子表格。你可能会把这些值作为参数（甚至可以写点代码直接从电子表格中抓取数据）。这都是很好的主意，但是一旦使用参数化测试，你将感受不到TDD的乐趣。

此外，你要记住TDD的目标是让测试以示例的方式记录系统行为，每个测试都经过恰当的命名来描述每个行为。虽然参数化测试能够满足这样的要求，但通常它们仅仅是测试而已。

4.6.2 测试中的注释

流行的测试工具都会带有一些代码示例，以作为发布包的一部分。这些代码中有许多注释详尽的测试。注释大概形式如下，带点学究气。

```
// Tests the c'tor that accepts a C string.
TEST(MyString, ConstructorFromCString)
```

这样的注释看起来让人感到不舒服，因为注释的意思和代码非常接近。对写代码的人来说，这浪费了精力和页面的空间；对于阅读代码的人而言，则是浪费阅读时间。

当然，注释不是测试工具的特性而是一个语言特性。在产品代码和测试代码中，最好的选择是尽量将注释化为更具表达力的代码。所剩的注释只回答类似下面的问题：为什么我会这样编写代码？

除了回答此类“为什么”，如果你要加注释来阐明测试的话，那就糟糕透顶了。测试应当清楚地阐明类的功能。你总是可以用一种无需使用阐述性注释的方法来重命名和组织测试（参见4.2.4节和7.3节）。

可以用一句话把这一点说得更清楚：不要用描述性的注释来总结测试，而是修改测试名称以达到描述效果。不要引导读者通过注释来理解测试。要整理测试中的步骤。

4.7 结束语

在学完本章中构建测试的方法和上一章中关于TDD的概念之后，你就可以应对更难的问题了。例如：怎样为一个必须与其他对象交互的对象写测试，尤其是这些对象对外部的依赖比较麻烦时？

第5章 测试替身

5.1 开场白

在前面三章中我们测试驱动开发了一个独立的类，也学习了TDD的所有基础内容。生活要是如此简单就好了！事实是，在真实的面向对象系统中对象必须协同工作。有时候，依赖合作对象会让TDD变得举步维艰：它们可能很慢、不稳定，或者帮不到你。

在本章中你将学习怎样使用测试替身（test double）[1]去解决这些难题。首先，你将学习怎样利用手工打造的测试替身去解除依赖。其次，你将学习怎样利用工具来简化测试替身的创建。你会学到多种设置代码的方法以便利用测试替身（又称作注入技术）。最后，你将了解使用测试替身给设计带来的影响，以及充分利用测试替身的策略。

5.2 依赖问题

通常，对象之间需要协同工作才能完成任务。一个对象通知另一个对象完成某件事，或者从另一个对象获取一些信息。如果对象A为了完成它的工作需要与对象B协同工作，那么我们就说对象A依赖对象B。

> **场景[2]：位置描述服务**
> 作为一个地图应用开发人员，我需要这样的服务，即它能基于给定的位置（经纬度）返回一行信息来描述离它最近的地方。

构建位置描述服务中一个重要的工作就是去调用一个外部API，这个API能接受一个位置信息，并返回位置数据。我找到了一个开放且免费的REST（Representational State Transfer，表述性

① 测试替身译法取自电影制作中的特型替身。例如，一个演员可能在武术动作方面不过关，这时候就要找一个专业的武术运动员帮助其完成动作。——译者注

② story。在敏捷开发中，一般使用一段描述性的话来表达需求，这叫作一个User Story，又可称为Scenario。本书中将之译为场景。——译者注

状态转移）服务，给定一个GET URL，它会以JSON格式返回位置数据。这个Nominatim搜索服务是Open MapQuest API的一部分[①]。

测试驱动开发位置描述服务会遇到一个难题。至少出于以下几点原因，对REST调用的依赖会成为一个问题。

- 通过一个HTTP来调用REST服务非常缓慢，这也导致测试的运行速度变慢（参见4.3节）。
- REST服务可能不是一直处于可用状态。
- REST服务返回的结果得不到保证。

为什么这些依赖会使得测试变得困难呢？首先，依赖一个慢速的协作对象会让测试慢得难以忍受。其次，依赖一个不稳定的服务（要么不可用，要么每次返回不同的结果）会导致测试间断性地失败。

完全出于*存在性*来讲，则会出现依赖。如果你没有支持发起HTTP调用的工具代码呢？在一个团队中，设计并实现一个正确的HTTP工具类或许是别人的工作内容。你没有时间去等他完成这个工作，也没时间自己去实现一个HTTP类。

如果你就是那个负责编写HTTP类的人呢？或许你想先探索一下位置描述服务的整体设计实现，然后再考虑HTTP工具类的具体实现细节。

5.3　测试替身

在上面提到的情形中，你可以使用*测试替身*来避免被这类问题挡住去路。测试替身起到代替的作用：一个doppelgänger（字面意思是double walker），它代替了实际产品代码中的类。HTTP给你带来麻烦了吗？如果答案是肯定的，那就创建HTTP的测试替身吧！测试替身的工作是满足测试所需。当客户提交一个GET请求至HTTP对象时，测试替身能够返回预先准备的响应。测试替身应该返回什么是由测试自己决定的。

想象一下你正负责构建一个服务，但是当下并不考虑单元测试（或许你计划写一个集成测试）。有下面几个可以重用的类供你选用。

- CurlHttp，它使用cURL[②]发起HTTP请求。这个类派生自纯虚基类Http，这个基类定义两个函数：`get()`和`initialize()`。客户端代码在调用`get()`前必须先调用`initialize()`。
- Address，一个包含几个字段的结构。

① 想了解更多关于此API的信息，可以访问http://open.mapquestapi.com/nominatim。维基百科提供了REST的概述，你也可以访问https://en.wikipedia.org/wiki/Representational_state_transfer。

② http://curl.haxx.se/libcurl/cplusplus/

- AddressExtractor，它借助JsonCpp[1]库从一个JSON[2]字符串中提取地址，并填写Address结构。

你的代码可能会是下面这样的：

```
CurlHttp http;
http.initialize();
auto jsonResponse = http.get(createGetRequestUrl(latitude, longitude));

AddressExtractor extractor;
auto address = extractor.addressFrom(jsonResponse);

return summaryDescription(address);
```

试想一下，你要为这段代码编写测试。然而这不太容易，因为CurlHttp类对外有依赖，这不是你所期望的。面对这个问题，许多开发人员会运行几个手动的测试，然后就继续编码了。

其实你可以做得更好，因为你是在进行测试驱动开发！这意味着只在让失败测试通过时才往系统中加入代码。但是要怎样写一个可以绕开CurlHttp类的对外依赖的测试呢？我们将在下一节中学习这个解决方案。

5.4 手动打造的测试替身

如果想要使用测试替身，那么就必须让它取代CurlHttp类的行为。C++提供了许多不同的方法，其中多态的使用频率最高。我们先来看一下CurlHttp类实现的Http接口。

c5/1/http.h

```
virtual ~Http() {}
virtual void initialize() = 0;
virtual std::string get(const std::string& url) const = 0;
```

你要做的就是在派生类中覆写虚函数，并在这个覆写中提供特别的行为来支持测试，然后将基类指针传递给地名描述服务。

现在来看一些代码。

c5/1/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, ReturnsDescriptionForValidLocation) {
   HttpStub httpStub;
   PlaceDescriptionService service{&httpStub};

   auto description = service.summaryDescription(ValidLatitude, ValidLongitude);
```

---

① http://www.json.org

② http://jsoncpp.sourceforge.net

```
    ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
}
```

我们在测试中创建了HttpStub类的一个实例。HttpStub正是测试替身的类型，它是一个派生自Http的类。我们在测试文件中直接定义了HttpStub，这样能方便地看到它的行为和使用它的测试代码。

c5/1/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
   void initialize() override {}
   std::string get(const std::string& url) const override {
      return "???";
   }
};
```

返回带有一些问号的字符串没什么用处。那该从`get()`中返回什么呢？由于外部Nominatim搜索服务返回的是一个JSON响应，因此我们也应该返回一个JSON响应，可以从这个响应中生成测试断言所期待的描述。

5

c5/2/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
   void initialize() override {}
   std::string get(const std::string& url) const override {
      return R"({ "address": {
         "road":"Drury Ln",
         "city":"Fountain",
         "state":"CO",
         "country":"US" }})";
   }
};
```

我是怎么想到这个JSON响应的呢？我用浏览器提交了一份真实的GET请求（Nominatim搜索服务API的文档会教你怎么做），然后截取了返回的结果。

在测试中，我们将HttpStub实例传递给了`PlaceDescriptionService`的构造函数。和原先的预想相比，我们正在改变设计。服务对象本身不创建私有的Http实例，相反，使用该服务对象的客户端需要自己创建一个Http实例，并把它传给服务对象。服务对象通过一个基类指针持有这个Http实例。

c5/2/PlaceDescriptionService.cpp

```
PlaceDescriptionService::PlaceDescriptionService(Http* http) : http_(http) {}
```

简单的多态赋予了我们所需的测试替身的灵活性。PlaceDescriptionService对象不知道它持有的Http实例是一个真正的实例，还是一个仅仅为了测试的实例。

一旦测试编译通过并运行失败，就可以编写`summaryDescription()`了。

c5/2/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   auto getRequestUrl = "";
   auto jsonResponse = http_->get(getRequestUrl);

   AddressExtractor extractor;
   auto address = extractor.addressFrom(jsonResponse);
   return address.road + ", " + address.city + ", " +
          address.state + ", " + address.country;
}
```

（幸运的是，其他人已经为我们构建了AddressExtractor。它能解析JSON响应，并填写Address结构体。）

当测试调用`summaryDescription()`时，对`get()`的调用实际作用在HttpStub实例上。`get()`返回的结果是预先硬编码的JSON字符串。返回硬编码值的测试替身叫作*存根*（stub）。类似地，我们也可以称`get()`为*存根方法*。

我们测试驱动开发了`summaryDescription()`。但是，请求的URL是什么？当正在接受测试的代码和一个协同对象交互时，需要保证给它传递一个正确的值。你怎么知道传给Http实例的URL是合法的呢？

实际上，在上述的代码中我们传给`get()`一个空字符串（即getRequestUrl），以便可以增量地开发。现在，我们需要编写能正确给`getRequestUrl`赋值的代码了。我们可以使用三角法，并为第二个位置添加一个断言（参考10.4.2节）。

更好的做法是在HttpStub中的`get()`中加入一个断言。

c5/3/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
   void initialize() override {}
   std::string get(const std::string& url) const override {
➤     verify(url);
      return R"({ "address": {
         "road":"Drury Ln",
         "city":"Fountain",
         "state":"CO",
         "country":"US" }})";
   }
   void verify(const string& url) const {
      auto expectedArgs(
         "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
         "lon=" + APlaceDescriptionService::ValidLongitude);
         ASSERT_THAT(url, EndsWith(expectedArgs));
   }
};
```

（为什么要为断言逻辑创建一个独立的`verify()`呢？这其实是Google Mock自身的一个限

制：你只能在返回void的函数中使用会导致致命错误的断言[1]。）

在调用get()时，存根实现可以确保参数符合预期。存根实现中的断言验证了URL最重要的方面：这个URL是否包含正确的纬度和经度。目前来说，测试是失败的，因为我们传给get()的是一个空字符串。我们可以做一下改动让测试通过，如下所示：

c5/3/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
➤  auto getRequestUrl = "lat=" + latitude + "&lon=" + longitude;
   auto jsonResponse = http_->get(getRequestUrl);
   // ...
}
```

由于当前的URL没有指定一个服务器或文档，因此它还不能很好地工作。我们可以增强verify()函数，让它传给get()一个完整的URL。

c5/4/PlaceDescriptionServiceTest.cpp

```
void verify(const string& url) const {
➤  string urlStart(
➤     "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&");
➤  string expected(urlStart +
      "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
      "lon=" + APlaceDescriptionService::ValidLongitude);
➤  ASSERT_THAT(url, Eq(expected));
}
```

一旦测试通过了，就要做一些重构。summaryDescription()方法缺乏整体感，测试和产品代码中构造“键–值”对的代码是重复的。修改后的代码如下：

c5/4/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   auto request = createGetRequestUrl(latitude, longitude);
   auto response = get(request);
   return summaryDescription(response);
}
string PlaceDescriptionService::summaryDescription(
      const string& response) const {
   AddressExtractor extractor;
   auto address = extractor.addressFrom(response);
   return address.summaryDescription();
}

string PlaceDescriptionService::get(const string& requestUrl) const {
   return http_->get(requestUrl);
}
```

---

① http://code.google.com/p/googletest/wiki/AdvancedGuide#Assertion_Placement

```
string PlaceDescriptionService::createGetRequestUrl(
      const string& latitude, const string& longitude) const {
   string server{"http://open.mapquestapi.com/"};
   string document{"nominatim/v1/reverse"};
   return server + document + "?" +
      keyValue("format", "json") + "&" +
      keyValue("lat", latitude) + "&" +
      keyValue("lon", longitude);
}
string PlaceDescriptionService::keyValue(
      const string& key, const string& value) const {
   return key + "=" + value;
}
```

那又该如何处理其他的重复代码呢？（你会问："还有哪些重复代码？"）测试中的文本和产品代码中的文本完全一样。应该消除这一重复吗？我们可以采取多种方法（参见7.4.6节）。

如果就此停止的话，目前的代码设计似乎也足够了。我们引入了一些可读性较高的函数。我们也需要做出一些改动。貌似可以重用`keyValue()`函数了。我们也能看出，可以迅速一般化现在的设计来支持第二个服务，因为PlaceDescriptionService中的一些结构是可以复用的。

但是，测试的设计还稍显不足。对于未参与代码编写的程序员来说，实在难以跟上。继续阅读吧！

5.5　在使用测试替身时提升测试的抽象程度

我们很容易创建难以阅读的测试，尤其是使用测试替身时，因为测试中模棱两可的信息增加了理解的难度。

因为ReturnDescriptionForValidLocation隐藏了相关的信息，所以理解起来比较困难。这和测试抽象理念相冲突（参见7.4节）。

c5/4/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, ReturnsDescriptionForValidLocation) {
   HttpStub httpStub;
   PlaceDescriptionService service{&httpStub};

   auto description = service.summaryDescription(ValidLatitude, ValidLongitude);

   ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
}
```

为什么获取的描述是一个在Colorado的Fountain城的地名呢？阅读测试的人必须要查看HttpStub实现中与此地址相关的JSON信息。

我们必须要重构测试，使它可以自包含。可以修改HttpStub的实现，让测试负责设定`get()`

方法的返回值。

c5/5/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
public:
   string returnResponse;
   void initialize() override {}
   std::string get(const std::string& url) const override {
      verify(url);
      return returnResponse;
   }

   void verify(const string& url) const {
      // ...
   }
};

TEST_F(APlaceDescriptionService, ReturnsDescriptionForValidLocation) {
   HttpStub httpStub;
   httpStub.returnResponse = R"({"address": {
                                    "road":"Drury Ln",
                                    "city":"Fountain",
                                    "state":"CO",
                                    "country":"US" }})";
   PlaceDescriptionService service{&httpStub};
   auto description = service.summaryDescription(ValidLatitude, ValidLongitude);
   ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
}
```

现在，阅读测试的人可以将摘要描述和HttpStub返回的JSON对象联系起来了。

类似地，我们也可以将URL验证移到测试中。

c5/6/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
public:
   string returnResponse;
   string expectedURL;
   void initialize() override {}
   std::string get(const std::string& url) const override {
      verify(url);
      return returnResponse;
   }
   void verify(const string& url) const {
      ASSERT_THAT(url, Eq(expectedURL));
   }
};

TEST_F(APlaceDescriptionService, ReturnsDescriptionForValidLocation) {
   HttpStub httpStub;
   httpStub.returnResponse = // ...
   string urlStart{
```

```
➤        "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&"};
➤     httpStub.expectedURL = urlStart +
➤        "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
➤        "lon=" + APlaceDescriptionService::ValidLongitude;
      PlaceDescriptionService service{&httpStub};

      auto description = service.summaryDescription(ValidLatitude, ValidLongitude);

      ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
   }
```

现在，测试变长了，但是能清晰地表达其意图。相比之下，我们将HttpStub削减至一个小类，它符合期望并返回所需的信息。由于它同时也验证了期望的信息，因此HttpStub从一个存根演化为了一个模拟对象（mock）。一个模拟对象是一个测试替身，它符合期望并且能够自我验证所期望的信息[①]。在我们的例子中，一个HttpStub对象验证了这样的事实：会有一个期望的URL传给HttpStub。

要测试驱动开发一个对诸如数据库和外部服务调用有依赖的系统，你需要多个模拟对象。如果这些对象仅仅是完成期望的任务，并返回期望的值，那它们长得都差不多。模拟对象工具可以帮助减轻定义测试替身的工作量。

5.6　使用模拟对象工具

模拟对象工具，如Google Mock、CppUMock或者Hippp Mocks，都能减轻定义模拟对象和设定期望的工作。你将在本节中学习如何使用Google Mock。

5.6.1　定义一个派生类

现在我们来重新测试驱动开发`summaryDescription()`。我们需要模拟HTTP方法：`get()`和`initialize()`。

c5/7/Http.h

```
virtual ~Http() {}
virtual void initialize() = 0;
virtual std::string get(const std::string& url) const = 0;
```

为了使用Google Mock自身的模拟对象，我们首先需要创建一个派生类用来声明所模拟的方法。Google Mock允许我们简洁地定义名为HttpStub的派生类。

c5/7/PlaceDescriptionServiceTest.cpp

```
class HttpStub: public Http {
public:
```

① *xUnit Test Patterns*[Mes07]

```
➤    MOCK_METHOD0(initialize, void());
➤    MOCK_CONST_METHOD1(get, string(const string&));
  };
```

你可以使用一些宏来声明模拟的方法。MOCK_CONST_METHOD1告诉Google Mock定义一个带有一个参数的const成员函数（MOCK_CONST_METHOD1末尾的1就是指一个参数）。宏的第一个参数，即get()，是成员函数的名称。你还要提供成员函数签明的其他部分（返回值和参数声明），即第二个宏参数给出的string(const string&)。

为了模拟initialize()，你可以使用MOCK_METHOD0，它创建了非const且无参数的函数声明。宏的第二个参数即void()，告诉Google Mock去声明一个不带参数且返回值为void的函数。

Google Mock还提供了模拟模板函数的支持，此外还可以指定调用约定[①]。

Google Mock把一个mock声明转为派生类中的一个成员函数。Google Mock还在幕后实现了这个函数。如果成员函数没有覆写基类成员函数，那么你就不会得到所需的行为。

（在C++11里，override关键字可以让编译器来验证是否正确地覆写了一个函数。但是，MOCK_METHOD宏还没有使用override关键字。访问https://code.google.com/p/googlemock/issues/detail?id=157来获取一个补丁，它可以修复Google Mock的这个问题。）

5.6.2 设立期望

我们决定删掉summaryDescription()的实现，然后重头测试驱动开发一次。这次，先缩小第一个测试的范围。我们先不去完全实现summaryDescription()，而是仅仅发送一个HTTP请求。我们使用一个新的测试名称来表达意图。

c5/7/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, MakesHttpRequestToObtainAddress) {
   HttpStub httpStub;
   string urlStart{
      "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&"};
   auto expectedURL = urlStart +
      "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
      "lon=" + APlaceDescriptionService::ValidLongitude;
➤  EXPECT_CALL(httpStub, get(expectedURL));
   PlaceDescriptionService service{&httpStub};

   service.summaryDescription(ValidLatitude, ValidLongitude);
}
```

在这个测试中我们使用EXPECT_CALL宏设立期望，让它作为测试Arrange（参见4.2.4节）步骤的一部分。这个宏配置Google Mock去验证给定一个参数去调用httpStub对象的get()得到的

① http://code.google.com/p/googlemock/wiki/V1_6_CheatSheet#Defining_a_Mock_Class

返回值是否与expectedURL吻合。

那么断言在哪呢？Google Mock会在模拟对象跳出作用域后才开始验证。看上去，我们的测试并没有遵循AAA（Arrange-Act-Assert），但是断言步骤仍然执行了，只不过是隐式地完成了。一些模拟工具需要显式地添加验证调用，这让断言步骤变得可见。

如果需要，也可以强制Google Mock在模拟对象跳出作用域前做验证。

```
Mock::VerifyAndClearExpectations(&httpStub);
```

虽然有些人喜欢加个显式的断言部分，但这不是必需的。你很快就会学会在设置模拟对象期望时如何去阅读测试。

我们只实现部分summaryDescription()，让测试编译通过就可以。

c5/7/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   return "";
}"
```

运行一下测试就会得到如下的失败信息：

```
Actual function call count doesn't match EXPECT_CALL(httpStub, get(expectedURL))...
         Expected: to be called once
           Actual: never called - unsatisfied and active
```

期望没有满足。直到测试结束为止，httpStub对象的get()也没有被调用。没能通过是好事！我们可以实现只够让测试通过的代码。

c5/8/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   string server{"http://open.mapquestapi.com/"};
   string document{"nominatim/v1/reverse"};
   string url = server + document + "?" +
                keyValue("format", "json") + "&" +
                keyValue("lat", latitude) + "&" +
                keyValue("lon", longitude);
   http_->get(url);
   return "";
}
```

我们可以写第二个测试让summaryDescription()变得具体化。

c5/9/PlaceDescriptionService.cpp

```
TEST_F(APlaceDescriptionService, FormatsRetrievedAddressIntoSummaryDescription) {
   HttpStub httpStub;
➤  EXPECT_CALL(httpStub, get(_))
➤     .WillOnce(Return(
```

```
➤         R"({ "address": {
➤              "road":"Drury Ln",
➤              "city":"Fountain",
➤              "state":"CO",
➤              "country":"US" }})"));
   PlaceDescriptionService service(&httpStub);
   auto description = service.summaryDescription(ValidLatitude, ValidLongitude);
   ASSERT_THAT(description, Eq("Drury Ln, Fountain, CO, US"));
}
```

测试中的`EXPECT_CALL()`调用会告诉Google Mock在调用`get()`时会返回什么。这里需要解释一下：对HttpStub对象调用`get()`会一次（就一次）返回一个特定的JSON字符串。

我们不在乎调用`get()`时会传递什么参数，因为在MakesHttpRequestToObtainAddress中已经验证了该行为。因此在`EXPECT_CALL()`中，我们使用了通配符`_`（即一个下划线，全称为`testing::_`）作为`get()`的参数。这个通配符可以让Google Mock匹配出所有对此函数的调用，不管参数是什么。

使用通配符可以剥离不相干的细节，这提升了测试的抽象程度。但是，不考虑URL参数意味着我们最好有其他测试（MakesHttpRequestToObtainAddress测试）来验证传入参数的合理性。换句话说，除非你的确不用考虑参数，或已经有了一个验证参数的测试，否则，不要使用通配符。

（通配符是Google Mock提供的众多匹配器中的一种。这些匹配器就是前面测试断言中所使用的。参考Google Mock的文档[1]获取一个全面的匹配器列表。）

让我们编写代码让测试通过。

c5/9/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   string server{"http://open.mapquestapi.com/"};
   string document{"nominatim/v1/reverse"};
   string url = server + document + "?" +
                keyValue("format", "json") + "&" +
                keyValue("lat", latitude) + "&" +
                keyValue("lon", longitude);
➤  auto response = http_->get(url);

➤  AddressExtractor extractor;
➤  auto address = extractor.addressFrom(response);
➤  return address.summaryDescription();
}
```

可以通过重构，让现有的实现与使用手工模拟对象情形得到的实现保持一致（参见code/ c5/10中的源代码）。

① http://code.google.com/p/googlemock/wiki/V1_6_CheatSheet#Matchers

5

5.6.3　松模拟和严模拟

细心的读者可能注意到了，我们在summaryDescription()的实现中并没有遵循CurlHttp接口。而且在调用get()前并没有调用initialize()。（不要忘记运行集成测试哦！）我们可以在MakesHttpRequestToObtainAddress测试中加入一个要求，确保initialize()被调用。

c5/11/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, MakesHttpRequestToObtainAddress) {
   HttpStub httpStub;
   string urlStart{
      "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&"};
   auto expectedURL = urlStart +
      "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
      "lon=" + APlaceDescriptionService::ValidLongitude;
➤  EXPECT_CALL(httpStub, initialize());
   EXPECT_CALL(httpStub, get(expectedURL));
   PlaceDescriptionService service{&httpStub};

   service.summaryDescription(ValidLatitude, ValidLongitude);
}
```

我们保持summaryDescription()中的总体策略不变，单独改动下get()来更新它的实现细节。

c5/11/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::get(const string& url) const {
➤  http_->initialize();
   return http_->get(url);
}
```

运行一下测试，我们会得到一个警告信息，不是来自于MakesHttpRequestToObtainAddress，而是另外一个测试FormatsRetrievedAddressIntoSummaryDescription。

```
GMOCK WARNING:
Uninteresting mock function call - returning directly.
    function call: initialize()
```

除了那些要求的交互，Google Mock还会捕捉所有与模拟对象的交互。这个警告意在帮助你了解没有预期的交互。

你的目标应该是零警告，或者将它们关闭，或者修复它们。放任警告信息不管将会产生更多的警告信息。这时，它们大多数都是无用的。忽略Google Mock产生的警告信息是不明智的，我们必须消除这些信息。

我们也有其他的选择，例如向FormatsRetrievedAddressIntoSummaryDescription中加入一个期望，但是这会向测试引入与测试目标无关的东西。我们要尽量避免与测试抽象理念相悖的解决方案。

每当遇到这种问题时，都要质疑一下现有的设计。虽然必须调用initialization，但可不可以换

到其他地方（比如在PlaceDescriptionService构造好后调用`initialization()`函数）？然而，将调用移到不同的地方并不能消除警告信息。

那测试的设计呢？当弃用手工打造的模拟解决方案转投Google Mock时，可以把一个测试拆分成两个。在一个测试中表达所有的东西，意味着在一个测试中为三个重要的事件（初始化、传入`get()`的消息、`get()`的返回值）设立期望。这是去掉警告信息的简单方法，但是，我们会得到一个杂乱的测试（参见7.3节来获取关于这个选择的讨论）。现在，让我们保持独立的测试。

我们也可以创建一个fixture辅助函数来为`initialize()`增加一个期望，然后返回一个HttpStub的实例。我们可以把这个函数命名为`createHttpStubExpectingInitialization()`。

Google Mock提供了更简单的解决方案（其他的模拟工具也提供了类似的解决方案）。NiceMock模板告诉Google Mock只追踪与设定期望的方法的交互。

c5/12/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, FormatsRetrievedAddressIntoSummaryDescription) {
➤   NiceMock<HttpStub> httpStub;
    EXPECT_CALL(httpStub, get(_))
       .WillOnce(Return(
       // ...
}
```

与之相反，使用StrickMock模板包装的模拟对象会将不关心的调用警告信息变为错误信息。

```
StrictMock<HttpStub> httpStub;
```

如果你使用NiceMock，那么就需要承担一定的风险。如果以后的代码调用Http接口的另一个方法，那么我们的测试将对此一无所知。不用习惯性地使用NiceMock，在需要的时候使用即可。如果需要经常使用的话，那么还是从设计上修正吧！

可以参考Google Mock的文档来了解更多关于NiceMock和StrickMock的信息[①]。

5.6.4 模拟对象中的顺序

虽然不太可能，但是如果不小心将对Http接口的`initialize()`和`get()`函数调用颠倒了，该怎么办呢？

c5/13/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::get(const string& url) const {
➤   auto response = http_->get(url);
➤   http_->initialize();
➤   return response;
}
```

① http://code.google.com/p/googlemock/wiki/CookBook#Nice_Mocks_and_Strict_Mocks

5

令人惊奇的是，测试也通过了！默认情形下，Google Mock不会验证满足期望的顺序。如果你在意顺序，可以让Google Mock（其他许多C++模拟工具也可以）去验证。最简单的方式是在测试的开始定义一个InSequence实例，然后确保接下来的EXPECT_CALLS按照期望的顺序出现。

c5/13/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, MakesHttpRequestToObtainAddress) {
➤  InSequence forceExpectationOrder;
   HttpStub httpStub;

   string urlStart{
      "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&"};

   auto expectedURL = urlStart +
      "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
      "lon=" + APlaceDescriptionService::ValidLongitude;
   EXPECT_CALL(httpStub, initialize());
   EXPECT_CALL(httpStub, get(expectedURL));
   PlaceDescriptionService service{&httpStub};

   service.summaryDescription(ValidLatitude, ValidLongitude);
}
```

如果你需要的话，Google Mock还提供了更精细的顺序控制。After()表明一个期望必须在另一个期望之后执行，这个期望才满足。Google Mock还容许你定义一个期望调用的顺序列表。可以参考Google Mock文档获取更多信息[1]。

5.6.5　巧妙的模拟工具特性

EXPECT_CALL宏支持许多修饰符。它的语法是：

```
EXPECT_CALL(mock_object, method(matchers))
    .With(multi_argument_matcher)  ?
    .Times(cardinality)            ?
    .InSequence(sequences)         *
    .After(expectations)           *
    .WillOnce(action)              *
    .WillRepeatedly(action)        ?
    .RetiresOnSaturation();        ?
```

（?和*代表每个修饰符的基数：?表示可以选用修饰符一次；*表示可以多次使用修饰符。）

最有用的修饰符是Times()，它可以让你指定一个方法被调用的次数。即使你知道一个方法会被调用多次，但不知道具体多少次，那么你也可以使用WillRepeatedly()。参考Google Mock

① http://code.google.com/p/googlemock/wiki/CheatSheet#Expectation_Order

文档获取更多信息[①]。

Google Mock是一个通用的单元测试工具，它支持几乎所有的模拟方式。例如，你可以使用Google Mock模拟出一个具有输出参数的函数。下面示例中的接口是在DifficultCollaborator类中定义的，它既能返回一个值，也能输出值到参数。

c5/13/OutParameterTest.cpp

```
virtual bool calculate(int* result);
```

你可以使用WillOnce()或WillRepeatedly()为Google Mock期望指定一个动作。大部分时候，这个动作是返回一个值。当一个函数不止需要返回一个值时，你就可以使用能将两个或多个动作组合起来的DoAll()。

c5/13/OutParameterTest.cpp

```
   DifficultCollaboratorMock difficult;
   Target calc;
   EXPECT_CALL(difficult, calculate(_))
➤     .WillOnce(DoAll(
➤        SetArgPointee<0>(3),
➤        Return(true)));

   auto result = calc.execute(&difficult);

   ASSERT_THAT(result, Eq(3));
```


这个测试中的另一个动作是SetArgPointee<0>(3)，它告诉Google Mock去设置被指者（即指针所指的）的第0个参数的值为3。

Google Mock为其他的动作提供了强有力的支持，包括抛出异常、设置变量、删除参数、调用函数或函数对象。可以参考Google Mock的文档获取完整列表[②]。

并不是因为可以使用这些特性就必须要用它们。大部分情况下使用之前学习到的Google Mock提供的基本机制就足够了。在测试驱动开发时，如果你经常要去使用怪异的模拟工具特性，那么请停下来检查一下你的设计。你所测的方法是不是做了过多的事情？为了使其不需要过度复杂的模拟，可以重新以另一种方式调整下设计吗？大多数情况下你是可以做到的。

当你尝试着为非测试驱动的、结构不良的系统编写测试并遇到问题时，或许需要使用模拟工具提供的更加强大的特性。我们会在第8章中讨论这类问题。

① http://code.google.com/p/googlemock/wiki/CheatSheet#Setting_Expectations,http://code.google.com/p/googlemock/wiki/ForDummies#Setting_Expectations

② http://code.google.com/p/googlemock/wiki/CheatSheet#Actions

5.6.6　排除模拟失败

你将会为正确地定义模拟对象而努力。说实在的：当我在为本书编写当前示例时，遇到了下面的错误信息，还真有点儿把我难住了。

```
Actual function call count doesn't match EXPECT_CALL(httpStub, get(expectedURL))...
         Expected: to be called once
           Actual: never called - unsatisfied and active
```

第一件需要确定的事情是，在产品代码中是否有合理的调用。（你有没有编写相应的代码？）第二件需要确定的事情是，你是否正确定义了模拟方法？如果不是很确定，那么可以在产品代码实现的第一行加一个cout语句，或者借助调试器来查明原因。

你有没有把要模拟的成员函数声明为虚函数？

你是不是把MOCK_METHOD()声明弄错了？在这里，所有的类型信息必须精确匹配，否则，模拟的方法将不会被覆写。同时你也要保证所有的const声明是一样的。

如果还是不行，那么就先排除关于参数匹配的担忧。为所有的参数和返回值使用通配符（testing::_）。如果测试通过，那么肯定有一个参数不能被Google Mock视为匹配。

我就犯过一个愚蠢的错误：不小心把const声明从URL参数前移除了。

5.6.7　一个还是两个测试

当使用手工创建的模拟对象时，我们最终只用了一个测试来验证最终的目标，即为一个位置生成概要信息描述。但是，在第二个示例中，我们却用了两个测试。哪个对呢？

从文档化公有行为（即客户所关心的行为）的角度来说，PlaceDescriptionService的唯一目标就是对给定的位置返回包含概要信息的字符串。对于测试阅读者来说，在测试中来描述这个行为更简单。客户并不关心summaryDescription()和Http协同对象是怎样交互的。这是一个实现细节（你或许会想到一个解决方案，这个方案中所有的地理信息都在本地）。创建第二个测试来阐明这个交互有意义吗？

当然有意义！TDD最重要的价值就在于帮助你开发和塑造系统的设计。与协同对象的交互是设计的关键。用测试来描述那些交互对于其他开发者来说很有价值。

拥有两个测试还会提供额外的好处。第一，一个模拟验证是一个断言。我们已经用一个断言来验证概要信息字符串了。将测试分为两个，与一个断言一个测试保持一致（参考7.3节）。第二，独立的测试更具可读性。由于在Google Mock中设立期望会导致很难界定设置断言的位置（这也和4.2.4节一致），因此我们为简化基于模拟的测试的努力是值得的。

5.7 让测试替身各就各位

在引入一个测试替身时需要做两件事。第一，编写测试替身。第二，在目标测试中使用测试替身的一个实例。这样的做法又称作*依赖注入*（Dependency Injection，DI）。

以PlaceDescriptionService为例，通过一个构造函数，我们将测试替身注入其中。在一些情况下，你会发现通过一个setter成员函数来注入测试替身更合适。这种方法又称作*构造函数注入*或*setter注入*。

也有很多其他让测试替身各就各位的方法，你可以根据自己的需求选用最适合的一种。

5.7.1 覆写工厂方法和覆写 Getter

为了使用覆写工厂方法[1]，必须修改产品代码，使其在任何需要一个协作类实例时都可使用工厂方法来获得。下面是在PlaceDescriptorService中达到此目的的一种改法：

5

c5/15/PlaceDescriptionService.h

```
#include <memory>
// ...
   virtual ~PlaceDescriptionService() {}
// ...
protected:
   virtual std::shared_ptr<Http> httpService() const;
```

c5/15/PlaceDescriptionService.cpp

```
#include "CurlHttp.h"
string PlaceDescriptionService::get(const string& url) const {
➤    auto http = httpService();
➤    http->initialize();
➤    return http->get(url);
}

➤ shared_ptr<Http> PlaceDescriptionService::httpService() const {
➤    return make_shared<CurlHttp>();
➤ }
```

相比于直接引用成员变量`http_`来实现与HTTP服务的交互，现在的代码调用一个保护成员函数`httpService()`来获取一个Http指针[2]。

在测试中，我们定义了PlaceDescriptionService的派生类。这个派生类的主要工作就是覆写返回一个Http实例的工厂方法（`httpService()`）。

① 工厂方法为设计模式中创建模式的一种，可以参考《设计模式》一书。——译者注
② 这里的指针是一个共享指针，是一种智能指针。——译者注

c5/15/PlaceDescriptionServiceTest.cpp

```
class PlaceDescriptionService_StubHttpService: public PlaceDescriptionService {
public:
   PlaceDescriptionService_StubHttpService(shared_ptr<HttpStub> httpStub)
      : httpStub_{httpStub} {}
   shared_ptr<Http> httpService() const override { return httpStub_; }
   shared_ptr<Http> httpStub_;
};
```

我们修改了测试来创建一个HttpStub的共享指针，并把它存在`PlaceDescriptionService_StubHttpService`的实例中。下面是修改后的MakesHttpRequestToObtainAddress：

c5/15/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService, MakesHttpRequestToObtainAddress) {
   InSequence forceExpectationOrder;
➤  shared_ptr<HttpStub> httpStub{new HttpStub};

   string urlStart{
      "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&"};

   auto expectedURL = urlStart +
      "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
      "lon=" + APlaceDescriptionService::ValidLongitude;
➤  EXPECT_CALL(*httpStub, initialize());
➤  EXPECT_CALL(*httpStub, get(expectedURL));
➤  PlaceDescriptionService_StubHttpService service{httpStub};

   service.summaryDescription(ValidLatitude, ValidLongitude);
}
```

覆写工厂方法展示了使用测试替身所带来的测试覆盖率漏洞。由于我们的测试覆写了产品代码中`httpService()`的实现，因此测试并没有使用实际产品代码中的这个函数。正如前面所说，要确保在集成测试中使用实际的服务！同时，不要在工厂方法中加入实际逻辑的代码，否则，未经测试的代码就会越来越多。工厂方法应当只返回协作类型的一个实例。

除了覆写工厂方法外，还可以使用覆写Getter方法。对我们的例子来说主要的不同点在于`httpSever()`是一个简单的getter，它只是返回一个指向已有的Http实例的成员变量。而在覆写工厂方法的实例中，`httpServer()`负责创建一个Http的实例。使用这种方法，测试保持不变。

c5/16/PlaceDescriptionService.h

```
class PlaceDescriptionService {
public:
➤  PlaceDescriptionService();
   virtual ~PlaceDescriptionService() {}
   std::string summaryDescription(
      const std::string& latitude, const std::string& longitude) const;

private:
   // ...
```

```
➤   std::shared_ptr<Http> http_;

protected:
   virtual std::shared_ptr<Http> httpService() const;
};
```

c5/16/PlaceDescriptionService.cpp

```
PlaceDescriptionService::PlaceDescriptionService()
   : http_{make_shared<CurlHttp>()} {}
// ...
shared_ptr<Http> PlaceDescriptionService::httpService() const {
➤   return http_;
}
```

使用得当的话，覆写工厂方法和覆写Getter会简单有效，特别是在应对遗留代码时（参见第8章）。但是，最好还是使用构造函数注入和setter注入。

5.7.2　使用工厂

工厂类是用来负责创建和返回实例的。如果你有一个HttpFactory，那么就可以在测试中告诉它返回一个HttpStub实例而非Http实例。如果使用工厂不合理的话，那么就不要使用这个技巧了。仅仅为了支持测试而引入工厂不是个好的选择。

下面是我们的工厂实现：

c5/18/HttpFactory.cpp

```
#include "HttpFactory.h"
#include "CurlHttp.h"
#include <memory>

using namespace std;

HttpFactory::HttpFactory() {
   reset();
}

shared_ptr<Http> HttpFactory::get() {
   return instance;
}

void HttpFactory::reset() {
   instance = make_shared<CurlHttp>();
}

void HttpFactory::setInstance(shared_ptr<Http> newInstance) {
   instance = newInstance;
}
```

在测试设置阶段，创建一个工厂，并在其中注入HttpStub。然后在接下来的`get()`中返回这个测试替身。

c5/18/PlaceDescriptionServiceTest.cpp

```
class APlaceDescriptionService: public Test {
public:
   static const string ValidLatitude;
   static const string ValidLongitude;

   shared_ptr<HttpStub> httpStub;
   shared_ptr<HttpFactory> factory;
   shared_ptr<PlaceDescriptionService> service;

   virtual void SetUp() override {
      factory = make_shared<HttpFactory>();
      service = make_shared<PlaceDescriptionService>(factory);
   }

   void TearDown() override {
      factory.reset();
      httpStub.reset();
   }
};

class APlaceDescriptionService_WithHttpMock: public APlaceDescriptionService {
public:
   void SetUp() override {
      APlaceDescriptionService::SetUp();
      httpStub = make_shared<HttpStub>();
      factory->setInstance(httpStub);
   }
};

TEST_F(APlaceDescriptionService_WithHttpMock, MakesHttpRequestToObtainAddress) {
   string urlStart{
      "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&"};
   auto expectedURL = urlStart +
      "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
      "lon=" + APlaceDescriptionService::ValidLongitude;
   EXPECT_CALL(*httpStub, initialize());
   EXPECT_CALL(*httpStub, get(expectedURL));
   service->summaryDescription(ValidLatitude, ValidLongitude);
}
```

我们修改`summaryDescription()`中的产品代码，以便从工厂中获取Http实例。

c5/18/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::get(const string& url) const {
➤  auto http = httpFactory_->get();
   http->initialize();
   return http->get(url);
}
```

因为我们是通过构造函数来传递工厂实例的，所以这种方法和构造函数注入略有不同，此外现在还多了一个间接层。

5.7.3 通过模板参数

有些注入技术比较巧妙。你也可以选择通过模板参数注入，它不需要客户程序传递一个协作类的实例。最好在已经使用模板的情形下使用模板参数注入。

我们把PlaceDescriptionService声明为一个模板，它有一个类型名称，即HTTP。在这个模板中加入一个成员变量，http_，其类型为HTTP。因为我们想让客户使用类名PlaceDescription Service，所以我们将模板类改名为PlaceDescriptionServiceTemplate。在定义模板之后，我们使用typedef来定义PlaceDescriptionService，它将产品类Http作为PlaceDescriptionServiceTemplate的模板参数[①]。下面是代码：

c5/19/PlaceDescriptionService.h

```
template<typename HTTP>
class PlaceDescriptionServiceTemplate {
public:
   // ...
   // 测试中的mock需要引用
   HTTP& http() {
      return http_;
   }
private:
   // ...
   std::string get(const std::string& url) {
      http_.initialize();
      return http_.get(url);
   }
   // ...
   HTTP http_;
};
class Http;
typedef PlaceDescriptionServiceTemplate<Http> PlaceDescriptionService;
```

我们在测试的fixture中声明的服务类型是以一个模拟类（即HttpStub）为模板参数的Place-DescriptionSeriveTemplate。

c5/19/PlaceDescriptionServiceTest.cpp

```
class APlaceDescriptionService_WithHttpMock: public APlaceDescriptionService {
public:
   PlaceDescriptionServiceTemplate<HttpStub> service;
};
```

测试不用为PlaceDescriptionService提供一个Mock的实例，而只需要提供Mock的类型。PlaceDescriptionService会创建这个类型的实例（作为成员变量http_）。因为Google Mock会验证和模板实例的交互，所以需要让测试访问它。为此，需要通过PlaceDescriptionServiceTemplate的

① PlaceDescriptionService是PlaceDescriptionServiceTemplate对于Http的一个特化版本。——译者注

访问函数`http()`来修改测试以便获得stub的实例。

c5/19/PlaceDescriptionServiceTest.cpp

```
TEST_F(APlaceDescriptionService_WithHttpMock, MakesHttpRequestToObtainAddress) {

   string urlStart{
      "http://open.mapquestapi.com/nominatim/v1/reverse?format=json&"};

   auto expectedURL = urlStart +
      "lat=" + APlaceDescriptionService::ValidLatitude + "&" +
      "lon=" + APlaceDescriptionService::ValidLongitude;
➤  EXPECT_CALL(service.http(), initialize());
➤  EXPECT_CALL(service.http(), get(expectedURL));

   service.summaryDescription(ValidLatitude, ValidLongitude);
}
```

你可以通过多种方法来使用模板参数引入Mock，有些会比这里的实现更加精巧（基于模板重定义模式，参见《修改代码的艺术》一书）。

5.7.4　注入工具

用于注入协作对象作为依赖对象的工具又称为*依赖注入工具*。有两个知名的C++例子，分别是Autumn框架[①]和Qt IoC容器[②]。Michael Feathers对当前C++中的依赖注入框架作出了评价：如果你想在C++中做依赖注入，那么就必须为要创建的类强加一些限制……你必须使它们继承自某个其他类，使用macro preregistraion或metaobject库[③]。首先，你需要掌握这里描述的手工注入技巧。其次，再去调研一下注入工具，来检查它们能否带来一些改善。依赖注入工具通常在完全支持反射机制的语言中更加有效。

5.8　设计会变化

或许你对使用测试替身的第一反应是，它会改变你的设计方法。这样可能会令你不安。但是不要担心，这是正常的反应。

5.8.1　内聚与耦合

在面对棘手的依赖时（如慢速的或不稳定的协同对象），最好的方法是将它们隔离成单独的类。当然，发送HTTP的请求也不是非常复杂。也许将此逻辑放在一个小而独立的Http类中是不

① http://code.google.com/p/autumnframework
② http://sourceforge.net/projects/qtioccontainer
③ http://michaelfeathers.typepad.com/michael_feathers_blog/2006/10/dependency_inje.html

值得的，但是选择这样的方式将带来重用的机会和更灵活的设计弹性（如可以用多态的方式来替换）。这样你在创建测试替身时也会多一些选择。

另一种方法是创建更加过程化、弱内聚的代码。让我们来看看在测试后行的世界里，一个PlaceDescriptionService的典型解决方案：

c5/17/PlaceDescriptionService.cpp

```
string PlaceDescriptionService::summaryDescription(
      const string& latitude, const string& longitude) const {
   // 通过API检索JSON响应
   response_ = "";
   auto url = createGetRequestUrl(latitude, longitude);
   curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
   curl_easy_perform(curl);
   curl_easy_cleanup(curl);

   // 解析JSON响应
   Value location;
   Reader reader;
   reader.parse(response_, location);
   auto jsonAddress = location.get("address", Value::null);

   // 填充从JSON获取的地址
   Address address;
   address.road = jsonAddress.get("road", "").asString();
   address.city = jsonAddress.get("hamlet", "").asString();
   address.state = jsonAddress.get("state", "").asString();
   address.country = jsonAddress.get("country", "").asString();

   return address.road + ", " + address.city + ", " +
          address.state + ", " + address.country;
}
```

上述实现的代码有二十来行，很方便读者阅读，尤其是引导性的注释。这是典型的后测试代码。虽然我们可以将其拆成多个更小的函数，就像稍早前的做法一样，但是开发人员不会这么做。测试后行的开发者不习惯定期做重构，通常，他们不需要使用快速的测试来让重构变得更快、更安全。这又怎么样呢？像这样的代码有什么不妥吗？我们可以在需要的时候去重构它。

从设计的角度看，这二十多行代码违背了单一责任原则——需要修改summaryDescription()的原因有多个：首先，这个函数与cURL紧密耦合；其次，这二十多行代码的函数算作冗长函数，想要完全理解它会花费很多时间。

像这样冗长的函数容易导致不必要的代码重复。例如，其他服务代码也有可能需要相同的cURL逻辑。开发人员往往会重复编写与cURL相关的代码，而不是尝试重用。重用始于一些可重用结构的独立，这些结构更容易被其他程序员识别。如果潜在的可以重用的代码深埋于冗长的函数中，那么重用将不会发生。

以这样的方式来开发系统会让你的代码数量加倍。

即便如此，你依然可以测试这二十来行代码。你也可以使用link substitution写一个快速的单元测试（参见8.9节），或者写一个能发起一个即时的REST服务调用的集成测试。但是由于此类测试会大一些，因此在单个的测试中就会有多个设置和验证动作（但总体而言，构建最初测试的工作量没有太大区别）。集成测试则相对慢一些，而且不稳定。

在现实世界里，比这二十来行代码差劲的代码比比皆是。当然，你的代码看起来会好一些，因为在践行TDD时，你会寻求高内聚、低耦合的设计。你会开始意识到灵活设计的好处，也会很快发现好的设计是怎样与测试和谐共存的，而且这些测试具有体量小，易于编写、阅读和维护的特点。

5.8.2　转嫁私有依赖

如果你不担忧测试，那么PlaceDescriptionService中的HTTP调用可以保有一个私有依赖，这意味着PlaceDescriptionService的客户端程序不会意识到HTTP调用的存在。但是，如果使用setter或构造函数注入，那么客户端程序就需要创建Http对象，并把它传给PlaceDescriptionService实例。这样，就将PlaceDescriptionService对HTTP的依赖转嫁给客户端程序了。

开发人员可能会担忧这样选择的后果。

> **提问：**setter或构造函数注入是不是违反了信息隐藏？
>
> **回答：**从客户端程序的角度来说，确实是违反了。但你可以使用其他依赖注入方法（参见5.7节）。即使某人利用了这些暴露出来的信息，也不大可能造成不好的影响。
>
> 此外，你也可以提供一个默认的实例。我们可以先配置PlaceDescription-Service使其包含一个CurlHttp实例，如果测试提供HttpStub实例，那么它将会被替代。而真正的产品客户端是不需要做什么改动的。
>
> **提问：**如果心怀恶意的开发者提供一个具有破坏性的Http实例呢？
>
> **回答：**如果产品的客户是团队之外的人，那么可以选择其他注入形式。如果担心团队内部的开发者有意利用注入点做些不好的事，那你将面临更大的问题。
>
> **提问：**我逐渐能够测试驱动开发了，但我担心仅仅为了测试的目的而改变我的设计方法。我的团队中的其他人可能也会这么觉得。
>
> **回答：**确切地了解软件能如期工作是改变设计方式的重要原因。你可以这样和同事讲：“我更关心代码是否如期工作。做出这么一个小的让步意味着我们能够更容易地测试代码，也会有更多的测试能够帮助我们更容易地打磨设计，我们也会对代码更有信心。所以你们能重新思考下我们的标准吗？”

5.9　使用测试替身的策略

使用测试替身和其他工具一样，最大的挑战不是学会怎么用它们，而是知道什么时候使用。本节将描述一些学派的意见，并给出合理使用测试替身的建议。

5.9.1　探索设计

现在假设AddressExtractor不存在。在测试驱动开发summaryDescription()时，你肯定会意识到需要一些逻辑，它能接受一个JSON格式的响应，并返回一个格式化的字符串。你可以在PlaceDescriptionService中全部自己实现这个逻辑。代码也不多（从现有的AddressExtractor代码看，也就十来行）。

一些程序员总是打着设计者的旗号，寻求有潜在重用性、更大灵活性和易于理解的代码的设计。为了遵守单一责任原则，或许可以将所需的逻辑拆分成两块：解析JSON格式的响应和格式化输出。

TDD允许你……不，它要求你在任何时候都要做出清醒的设计选择。实现summary-Descritpion()的方法有无数种。你可以使用TDD来帮助探索这些设计。通常，这从一段可以工作的代码开始，然后重构至合理的解决方案。

5

你也可以先写一个用来描述summaryDescription()应该怎样和外部协作者交互的测试。这个协作者的工作是得到一个JSON格式的响应，并返回相应的地址数据结构。目前而言，我们可以忽略实现这个协作者的细节，先集中使用mock来测试驱动开发summaryDescription()，就像我们为和Http对象交互所做的一样。

当以这种方式测试驱动开发时，可以通过引入mock来替代缺失的协作者行为。要根据客户端的需求为协作者设计接口。

在某一时刻，你或其他人将会实现这个协作者。这时你可以作出以下选择：移除mock，以便待测试的代码使用产品级的协作者；保留mock。

也许你已经作好了选择。如果协作者引入麻烦的依赖，那么就需要保留mock。否则，移除mock会降低测试的复杂度。但是，你也许选择保留它，特别是需要用它来描述与协作者的交互式设计中的重要方面。

或许最好的指导方针需要考虑维护和理解测试所需的精力。如果没有mock，事情可能会简单些，但不总是这样的。mock可能需要大量的代码来初始化一些协作者，这也会增加维护测试的成本。

太多的模拟？

在2003年的7个月里，我在一个大型Java开发团队中担任程序员并践行XP。刚到的时候，我看到大量的单元测试时很激动，但是当看到测试的质量或产品

代码的质量时就没那么兴奋了。这两部分代码虽说不是那么糟糕，但是我发现了许多重复的代码，冗长的方法和测试，以及大量使用的mock。然而，系统却可以正常工作，也没暴露出过多缺陷。

几个月过后，当我尝试将系统用户从11个增加到50多个时，出现了性能问题。所有迹象表明问题出在没有很好地利用中间件框架。后来，一个收费颇高的中间件框架专家和我们的团队紧密合作来重整代码。

可以利用控制器级别的mock查看是否按照一定的顺序调用方法，以便验证事件发生的顺序。（更糟糕的是，他们使用的工具要求以字符串的方式给出函数的期望。重命名一个方法意味着必须记住去更新mock所用的字符串。）但是，这些方法没那么抽象。当优化团队开始重整代码以提高性能时，通常要改变底层给定消息流的设计。这就需要把方法移来移去、重命名方法、将两个方法合二为一、删掉一些方法，等等。每当他们做这样的改动……哦！测试失败了，有时候一次改动会导致十来个代码的失败。

由于测试紧密地和目标实现耦合在一起，因此重整代码会变得非常缓慢。副总裁和其他反对者开始抱怨。所有的程序员和TDD本身也遭受质疑。

如果测试严重依赖具体实现（以特定的顺序调用特定的函数）的话，那么就会导致严重的问题。事后看来，更大的问题来自于不充分的系统设计。如果去掉测试中的重复，可能会让我们更加轻松。另一个严重的问题是没有合适的性能和伸缩性测试。

5.9.2　mock 流派

把TDD用于设计探索的开发者属于伦敦流派。这一流派的的创始人包括Tim Mackinnon、Seteve Freeman和Philip Craig，他们首先发表了关于mock的论文（Endo-Testing: Unit Testing with Mock Objects [MFC01]）。由Freeman和Nat Pryce合著并获得高度好评的*Growing Object-Oriented Software, Guided by Tests [FP09]*中集中讨论了使用TDD来开发系统。

由于伦敦学派强调对象间的交互，因此它也促进了Tell-Don't-Ask的理念的发展。在面向对象的系统中，你（一个客户端对象）向一个对象发送消息来告诉它要完成的任务，并让它自主完成。你不需要查询对象信息然后完成可能在这个对象职责范围内的任务。Tell-Don't-Ask催生了更松耦合的设计。

经典流派（有时又称作Cleveland流派）强调通过查看状态来验证行为。Ken Beck的《测试驱动开发（中文版）》几乎完全集中于TDD方式。这一学派的开发者在依赖关系引起问题时才引入mock。

引入mock会让测试对其测试目标内部的实现产生依赖。如果你使用一个工具，同样也会依赖这个工具。如果操作期间粗心的话，那么这两种依赖会让设计变得不灵活，而且测试也不稳定。

预防这一问题的最好方法是：隔离并最小化它们。

与此同时，开发者也要为使用mock带来的额外复杂度买单（我就曾花费了数分钟来找出一个mock声明错误）。

作为专业人员，你应该对这两种流派的方法有所了解。虽然你可能会选择遵循其中的一个流派，但依然可以将伦敦流派和经典流派的元素纳入到TDD实践中。

5.9.3 明智地使用测试替身

如果你要彻头彻尾地测试驱动开发一个带有快速测试的系统，这其中的大部分系统都需要使用测试替身。在使用测试替身时，可以参考下面的建议。

> **重新思考设计**。你是为了简化依赖对象的创建而使用mock的吗？如果是的话，那么重新修改依赖结构。你是不是在多个地方为同一个东西使用了mock？如果是的话，那么重新设计来消除这样的重复。
>
> **意识到单元测试覆盖率上的让步**。一个测试替身代表了系统测试覆盖率的漏洞。因为测试替身提供的逻辑正是单元测试所不能覆盖的，所以一定要确保其他测试覆盖到这部分逻辑。
>
> **重构测试**。不能让对第三方工具的依赖成为问题。使用随意的方法会导致mock的大量增加，也会导致大量的重复和复杂难懂的测试。要像重构产品代码那样去重构测试！把期望声明封装进一个公共的辅助函数能够提高抽象、降低依赖度、减少重复代码。当你日后想升级到一个更新的、比Google mock更好用的工具的时候，就不再需要做非常大的改动。
>
> **质疑以过度复杂的方式使用测试替身**。如果你身陷mock，那可能是因为你尝试了过度测试或你的设计有缺陷。这时使用多级的mock通常能解决问题。而使用Fake（参见5.10节）往往会导致更多的问题。如果遇到了问题，那么就要将测试分解为多个小的测试来简化问题。同样也要检测一下所测代码是否可以拆解。
>
> **表达力胜于功能**。选择mock工具是因为它能帮助你创建高度抽象的测试，这些测试可以文档化系统行为和设计，而不仅仅是因为它有很酷的功能，可以做精巧和深奥的事情。除非必要，否则不要使用这些精巧深奥的功能。

5.10 其他关于测试替身的主题

我们将在最后这一节中学习一些使用测试替身的零散知识，包括广泛接受的术语、该在哪定义它们、该不该模拟具体的类，还有它们对性能的影响。

5.10.1　怎么称呼它们

到目前为止，本章中使用的术语有测试替身、mock和Stub。TDD社区中的大部分人都接受了这些术语的定义，还有一些对你或许有用的术语。你会经常听到用mock来取代测试替身。大部分时候，这是合适的，因为大部分开发者都使用mock工具。但是，如果想要更顺畅地沟通的话，那么就要依据所处的环境来使用最合适的术语。*xUnit Test Patterns* [Mes07]为这些定义提供了权威解释。

> **测试替身**：为测试而模拟产品代码的代码。
> Stub：一个返回硬编码值的测试替身。
> Spy：一个保存接受信息以便日后验证的测试替身。
> mock：一个基于期望自我验证的测试替身。
> Fake：提供产品类轻量级实现的测试替身。

我们为`get()`手工打造的测试替身既是一个Stub也是一个Spy。说它是Spy，是因为它验证收到的URL包含一个正确的HTTP GET请求的URL。说它是Stub，是因为它返回了硬编码的JSON文本。之后，我们用Google Mock将之实现为mock，用来捕获期望，并自动验证期望是否满足。

内存数据库[①]是Fake的典型例子。因为和基于文件系统的数据库交互本身很慢，所以许多团队实现了一个测试替身，用来模拟大部分与数据库的交互。其底层的实现一般是基于哈希的数据结构，它提供了简单且快速基于键值的查询。

使用Fake带来的挑战是它成为一个first-rate类，通常实现也会变得复杂，这样可能自身就有缺陷。例如，如果使用数据库Fake的话，就必须基于哈希的实现正确地完成数据库的语义交互。这是可以做到的，但也容易犯错。

尽量不要使用Fake。不然你肯定会花费半个下午才能找到由Fake自身的小缺陷导致的问题（我就有这样的经历）。测试集的存在是为了简化和加速开发，而不是自找麻烦，浪费时间。

如果你坚持使用Fake，那么就要保证Fake自身能够通过单元测试。为Fake开发的测试需要证明其行为和模拟的对象一致。

5.10.2　测试替身该放在哪

一开始在相同的测试文件中定义测试替身，以便开发者看到。当多个fixture使用同一个测试替身时，再将声明移植到单独的头文件中。当不再需要查看测试替身时，应该把它们从视野中移除。

---

① in-memory database将数据库的所有数据放在内存中，可以加速数据库的访问，这样就加快测试的执行速度。
——译者注

你需要记住的是，修改产品接口会导致使用测试替身的测试失败。如果你们同在一个团队，并遵守集体代码所有权准则（每个人都有权修改任何代码），那么修改产品代码的程序员就要负责运行所有的测试并修复所有的问题。在其他情况下，最好发个消息清楚地传达改动及其影响。

5.10.3 虚函数表和性能

引入测试替身是为了测试驱动开发一个有复杂依赖关系的类。许多创建测试替身的技术都需要创建派生类来覆写虚成员函数。如果之前的产品类没有虚函数，那么现在会有，并且会有一个虚函数表。虚函数表带来额外的间接性是有开销的。

引入虚函数表会引起对性能的担忧，因为C++需要在虚函数表中做额外的查询（而不是简单地调用一个函数）。而且，编译器不能内联一个虚函数。

但是，在大多数情况下，虚函数表带来的性能影响可以忽略不计甚至没有影响。在特定情形下，编译器能够优化这些开销。大部分时候，你会想要一个更好的、多态的设计。

然而，如果你必须大规模地调用模拟的产品函数，那么就需要先得到一些性能数据。如果性能降到不可接受的程度，就要考虑不同的模拟方式（或许基于模板的方案），重新设计（可能的话，通过优化其他地方来补偿性能损失），或者引入集成测试来弥补单元测试的不足（参见10.2节）。

5.10.4 模拟具体的类

在前面的示例中，我们通过实现一个纯虚的Http接口来创建一个mock。很多系统主要由具体的类构成，因此没有多少此类接口。从设计的角度讲，使用接口是将系统中的一部分和另一部分隔离的方式。依赖倒置原则（Dependency Inversion Principle，DIP）[①]提倡让客户端依赖抽象的接口而非具体的实现，从而达到消除依赖的目的。以纯虚类的方式引入这种抽象，能够加快编译并隔离复杂性。更重要的是，它们能让测试变得简单。

如果是必须的话，你可以创建一个派生自一个具体类的mock。问题是产生的类混合了产品代码和模拟的行为，这又被称为*部分模拟*。首先，部分模拟通常能告诉你所模拟的类过大，如果你只需要其中的一部分而不是全部，那么就可以按照这些边界将类拆分成两个。其次，使用部分模拟很可能使你陷入麻烦，以至于很快陷入模拟地狱。

如果你想通过定义一个派生类来直接模拟CurlHttp，那么你就会默认调用它的析构函数。这样做有什么问题吗？或许会有问题，因为它正好直接与cURL库交互。也许你不希望你的测试是这样的。在一些情况下，你也可能遇到诡秘的问题：“啊！我原以为代码此时会与模拟的方法交互，但是看起来它与真正的方法交互了。”你肯定不想经常浪费时间来解决此类问题。

---

① 《敏捷软件开发》

如果你使用诸如部分模拟此类难以驾驭的工具，那么你的设计正散发着坏味。举例来说，一个干净的设计会采用具体类继承一个接口的方式。这样，测试也不再需要部分模拟了，为这个接口创建一个测试替身即可。

5.11　结束语

本章中介绍了一些可以在测试驱动开发时使用的技巧，使用这些技巧可以打破对协作者的依赖。你需要查看模拟工具的文档来完全理解其额外的特性及细微差别。另外，你现在已经对TDD中构建产品级系统所需的核心机制有了基本的理解。

然而，这并不意味着你可以停止阅读，特别是想获得长远的成功的情况下。我们才刚刚开始学到有用的东西。既然可以测试驱动开发所有的代码，那么怎样利用它才能让设计保持干净和简约呢？怎样确保测试也保持干净和简单呢？我们在接下来的两章将集中讨论如何提升产品代码和测试的设计，让维护开销一直处于最小。

第 6 章 增量设计

6

6.1 开场白

从机制到推荐的实践，再到应对依赖的各种技巧，你已经学习了TDD的核心基础，一路上也在不断重构[①]，以每个改动增量地打磨代码，但是什么时候才能结束呢？

使用TDD的主要原因是，能够以可承受的、稳定的维护成本来添加或修改功能特性。通过在改动系统时允许持续地改进设计，TDD提供了这种支持。TDD实践产生的测试证明，系统的逻辑是按预期的方式工作的，你可以在加入新代码后整理代码。这对我们来说意义重大。没有TDD，你就不能得到快速的反馈，也不能安全、容易地做增量的代码改动。如果不能增量地修改代码，你的代码库会逐渐退化。

在本章中，你将学到重构过程中需要做的事情。我们将主要讨论Kent Beck提出的简单设计理念（参见《解析极限编程：拥抱变化》），以及可以保持代码整洁的一系列重要规则。

6.2 简单设计

如果你对设计一无所知，那么在使用TDD时需要考虑以下三条简单规则。

- 确保代码具备很强的可读性和表达力。
- 在和第一条规则不冲突的情况下，消除所有的重复。
- 不要向系统引入不必要的复杂性。避免猜测性的结构关系（“我只知道总有一天我们不得不支持多对多的关系”）和不能增强系统表达力的抽象。

虽然最后一条规则很重要，但是其他两个优先级更高。换句话说，如果新的成员函数或类能够提升表达力的话，那么就加入这一抽象好了。

---

① XP（eXtream Programming）的创始人及重构的倡导者Kent Beck对重构做了一个精彩的总结：了解并实践重构技术只是你登堂入室的一块敲门砖。在重构的历程中，如果哪天你可以自信满满地停止重构，那才是真正的得道。——译者注

恪守这些规则能够让你的系统可维护性变得特别好。

6.2.1　重复代码的代价

在代码示例中，我们对消除重复代码关注颇多。但这是为什么呢？

随着时间的推移，重复代码或许是维护代码库的最大开销。试想一下，你的系统需要用两行代码来审查一些重要的事件。当然，这两行代码可以放在一个辅助性的成员函数中，但只有两行代码，对吧？当需要增加一个待审查的事件时，最简单的方式就是找到另一个审查点，然后复制、粘贴这两行代码。

两行重复代码听起来没那么糟，但如果这两行代码在你的整个代码库中重复了100次，情况会怎样？再想想，如果审查的需求要修改一下，需要加入第三行代码。噢喔！你必须找出100个需要修改的地方，加上这行代码，然后重新运行所有测试，这样的成本远比只改一个地方要高得多。如果其中一处的重复代码稍有不同呢？你必须花费额外的时间进行分析，来断定这个重复代码的变体是不是有意的。如果你只找到99个重复点却漏掉了第100个重复点呢？那么你就发布了有缺陷的代码。

大多数开发者懒于创建新的成员函数，因为怀疑这会降低性能，所以他们有时甚至会拒绝这样做。最终，他们只是为自己创造了更多的未来工作量。

这不是导致重复代码的唯一途径。试想你被告知要添加一个已有特性的变体。这需要你改动六行代码，如果条件满足，就执行它们，否则执行已有的六行代码。

你发现已有的特性是一个未经测试的、约200行代码的成员函数。正确的设计方式是抽取195行左右的公共代码行，并允许用扩展的方式引入变体。或许模板方法或策略[①]设计模式可为此方案提供基础。

大部分程序员没能正确实现，不是因为他们不知道怎么做，而是因为他们懒得做、不敢做。“如果改变已有的代码破坏了功能，我会因此受到责备的，我不该一开始就乱来。”这样做更容易些：复制200行代码，修改好后继续工作。

由于对重复的自然倾向，大部分大型系统的代码远远多于实际需要的代码。这些额外的代码大大地增加了维护成本和风险。

将增量重构作为TDD环节的一部分可以避免系统级的退化。

6.2.2　投资管理器

让我们看看Beck的简单设计理念在开发小型子系统时是怎样发挥作用的。

---

① 可参考《设计模式》中的行为模式一章。——译者注

场景：投资管理器

投资人想要跟踪股票买卖记录，并将此作为金融分析的基础。

经过努力，我们在测试驱动投资管理器中得到了下面代码。（注意，这个示例的大部分代码是在幕后构建的。我将仅展示与设计讨论相关的部分。可以通过下载的源代码来查看完整的代码。）

c6/1/PortfolioTest.cpp

```
#include "gmock/gmock.h"
#include "Portfolio.h"

using namespace ::testing;

class APortfolio: public Test {
public:
   Portfolio portfolio_;
};

TEST_F(APortfolio, IsEmptyWhenCreated) {
   ASSERT_TRUE(portfolio_.IsEmpty());
}

TEST_F(APortfolio, IsNotEmptyAfterPurchase) {
   portfolio_.Purchase("IBM", 1);

   ASSERT_FALSE(portfolio_.IsEmpty());
}
TEST_F(APortfolio, AnswersZeroForShareCountOfUnpurchasedSymbol) {
   ASSERT_THAT(portfolio_.ShareCount("AAPL"), Eq(0u));
}

TEST_F(APortfolio, AnswersShareCountForPurchasedSymbol) {
   portfolio_.Purchase("IBM", 2);
   ASSERT_THAT(portfolio_.ShareCount("IBM"), Eq(2u));
}
```

c6/1/Portfolio.h

```
#ifndef Portfolio_h
#define Portfolio_h

#include <string>

class Portfolio {
public:
   Portfolio();
   bool IsEmpty() const;
   void Purchase(const std::string& symbol, unsigned int shareCount);
   unsigned int ShareCount(const std::string& symbol) const;

private:
   bool isEmpty_;
   unsigned int shareCount_;
```

```
};

#endif
```

c6/1/Portfolio.cpp

```
#include "Portfolio.h"
using namespace std;
Portfolio::Portfolio()
   : isEmpty_{true}
   , shareCount_{0u} {
}
bool Portfolio::IsEmpty() const {
   return isEmpty_;
}

void Portfolio::Purchase(const string& symbol, unsigned int shareCount) {
   isEmpty_ = false;
   shareCount_ = shareCount;
}

unsigned int Portfolio::ShareCount(const string& symbol) const {
   return shareCount_;
}
```

你能够通过阅读测试知道Portfolio类的功能吗？你应当养成习惯，通过阅读测试名称来了解一个类的设计意图。测试就是你的理解途径。

6.2.3　投资管理器中的简单重复

测试和产品代码中都存在重复代码。字符串"IBM"在两个测试中就重复了3次：在IsNotEmptyAfterPurchase中出现一次，在AnswersShareCountForPurchasedSymbol中出现2次。将字符串提取为常量能使其更易读，同时降低了日后在代码中拼错的风险，还使得编写新的测试更简单。此外，如果要变更IBM的符号，在一个地方就可以完成修改。

c6/2/PortfolioTest.cpp

```
#include "gmock/gmock.h"
#include "Portfolio.h"

using namespace ::testing;
using namespace std;

class APortfolio: public Test {
public:
➤   static const string IBM;
    Portfolio portfolio_;
};
➤ const string APortfolio::IBM("IBM");
// ...
TEST_F(APortfolio, IsEmptyWhenCreated) {
```

```
   ASSERT_TRUE(portfolio_.IsEmpty());
}

TEST_F(APortfolio, IsNotEmptyAfterPurchase) {
   portfolio_.Purchase(IBM, 1);
   ASSERT_FALSE(portfolio_.IsEmpty());
}
// ...
TEST_F(APortfolio, AnswersZeroForShareCountOfUnpurchasedSymbol) {
   ASSERT_THAT(portfolio_.ShareCount("AAPL"), Eq(0u));
}
TEST_F(APortfolio, AnswersShareCountForPurchasedSymbol) {
   portfolio_.Purchase(IBM, 2);

   ASSERT_THAT(portfolio_.ShareCount(IBM), Eq(2u));
}

TEST_F(APortfolio, ThrowsOnPurchaseOfZeroShares) {
   ASSERT_THROW(portfolio_.Purchase(IBM, 0), InvalidPurchaseException);
}
```

这是不是就意味着遇到相同的字符串就要提取为变量呢？假设AnswersShareCountForPurchasedSymbol只是一个需要字符串`"IBM"`的测试。创建一个局部变量`IBM`会赋予我们之前所说的好处。但是这种情况下，变量的值似乎意义并不大。我们可以轻易地看出这个测试中使用字符串的地方，所以这是无足轻重的。如果需要的话，也可以安全地改动它们。

6

设计往往就是作出判断。尽力恪守本章开头提出的设计原则可以更好地理解你的系统是怎样从中获益的。有了这样的经验后，当遇到需要作出判断的代码时，你就能理解摒弃设计规则的影响了。

阅读以下产品代码，看看你能不能找出重复代码：

c6/1/Portfolio.cpp

```
#include "Portfolio.h"
using namespace std;
Portfolio::Portfolio()
   : isEmpty_{true}
   , shareCount_{0u} {
}
bool Portfolio::IsEmpty() const {
   return isEmpty_;
}

void Portfolio::Purchase(const string& symbol, unsigned int shareCount) {
   isEmpty_ = false;
   shareCount_ = shareCount;
}

unsigned int Portfolio::ShareCount(const string& symbol) const {
   return shareCount_;
}
```

从视觉角度来看，代码并没有明显的行与行（或表达式与表达式）的重复。但其中存在算法级的重复。成员函数IsEmpty()返回一个布尔量，这个布尔量会在Purchase()被调用时更改。但是，空的概念却直接绑定了股票数目，股票数目会在调用Purchase()时被赋值。通过去除isEmpty_变量，让IsEmpty()查询股票的数量，我们可以消除这一概念重复。

c6/2/Portfolio.cpp

```
bool Portfolio::IsEmpty() const {
➤   return 0 == shareCount_;
}
```

（是的，通过查询股票数量来决定空不是很方便，但目前来说是正确的，换句话说，以增量的思维来开发。知道自己在编写临时代码可能会激发一些有趣的想法，从而促进新测试的产生。实现存在的问题是，如果某人购买了0股某股票，portfolio会返回空。我们对于空的定义是，portfolio是否包含任何股票。所以，这是空吗？或者，我们是不是应该不允许这笔买入？为了能继续推进，我们选择后者，写一个名为ThrowsOnPurchaseOfZeroShares的测试。）

算法的重复（解决同一问题的不同方法或问题的不同部分）会随系统增长演变为重大问题。通常来说，随着对一个实现的改动未能编写进其他实现，重复代码会演化为不经意的变体。

6.2.4　我们真的能坚持增量方法吗

经过一段时间的编码后，我们得到了如下的一些测试（目前仅是一些测试名称，因为你应该能想象出这些测试是做什么的）……

c6/3/PortfolioTest.cpp

```
TEST_F(APortfolio, IsEmptyWhenCreated) {
TEST_F(APortfolio, IsNotEmptyAfterPurchase) {
TEST_F(APortfolio, AnswersZeroForShareCountOfUnpurchasedSymbol) {
TEST_F(APortfolio, AnswersShareCountForPurchasedSymbol) {
TEST_F(APortfolio, ThrowsOnPurchaseOfZeroShares) {
TEST_F(APortfolio, AnswersShareCountForAppropriateSymbol) {
TEST_F(APortfolio, ShareCountReflectsAccumulatedPurchasesOfSameSymbol) {
TEST_F(APortfolio, ReducesShareCountOfSymbolOnSell)  {
TEST_F(APortfolio, ThrowsWhenSellingMoreSharesThanPurchased) {
```

类的实现如下：

c6/3/Portfolio.cpp

```
#include "Portfolio.h"
using namespace std;
bool Portfolio::IsEmpty() const {
   return 0 == holdings_.size();
}
void Portfolio::Purchase(const string& symbol, unsigned int shareCount) {
   if (0 == shareCount) throw InvalidPurchaseException();
   holdings_[symbol] = shareCount + ShareCount(symbol);
}
```

```
void Portfolio::Sell(const std::string& symbol, unsigned int shareCount) {
   if (shareCount > ShareCount(symbol)) throw InvalidSellException();
   holdings_[symbol] = ShareCount(symbol) - shareCount;
}

unsigned int Portfolio::ShareCount(const string& symbol) const {
   auto it = holdings_.find(symbol);
   if (it == holdings_.end()) return 0;
   return it->second;
}
```

这个时候，我们被告知一个新的场景。

场景：显示买入历史记录

投资者想看一下特定股票的购买记录，每个记录要显示购买的日期及数量。

就现有的实现而言，这个场景将了我们一军，因为我们没有跟踪单笔买入，也没有记录购买日期。这也是许多开发者质疑TDD的地方。如果多花些时间做一些前期的需求分析，那么我们就会知道需要跟踪买入日期。这样的话，我们的初期设计就可能会纳入这个需求。

这个场景所产生的改动似乎正好，10多分钟一次。我们必须定义好表示买入的数据结构、改变方法的参数列表、从客户端代码提供日期（目前而言，仅仅是我们的测试）、正确地填写数据结构，并存储数据。

不，先不要这样做……至少不要一口气做完。让我们看看能否增量地进行，每几分钟就寻求一下正面的反馈。其中一种做法是作出假设。让我们先创建一个做出一笔买入的测试，然后验证相应的买入是否在购买记录中。假设买入总是在一个指定的日期做出，因为可以不给`Purchase()`传递日期，这使得目前的任务更简单。

c6/4/PortfolioTest.cpp

```
TEST_F(APortfolio, AnswersThePurchaseRecordForASinglePurchase) {
   portfolio_.Purchase(SAMSUNG, 5);
   auto purchases = portfolio_.Purchases(SAMSUNG);

   auto purchase = purchases[0];
   ASSERT_THAT(purchase.ShareCount, Eq(5u));
   ASSERT_THAT(purchase.Date, Eq(Portfolio::FIXED_PURCHASE_DATE));
}
```

为了让测试通过，甚至不需要将买入记录和`holdings_`数据结构相关联。因为目前的假设只考虑单次买入，所以可以定义一个“全局的”买入记录集合。

c6/4/Portfolio.h

```
struct PurchaseRecord {
   PurchaseRecord(unsigned int shareCount, const boost::gregorian::date& date)
   : ShareCount(shareCount)
   , Date(date) {
   }
```

```
   unsigned int ShareCount;
   boost::gregorian::date Date;
};

class Portfolio {
public:
➤  static const boost::gregorian::date FIXED_PURCHASE_DATE;

   bool IsEmpty() const;

   void Purchase(const std::string& symbol, unsigned int shareCount);
   void Sell(const std::string& symbol, unsigned int shareCount);

   unsigned int ShareCount(const std::string& symbol) const;
➤  std::vector<PurchaseRecord> Purchases(const std::string& symbol) const;

private:
   std::unordered_map<std::string, unsigned int> holdings_;
➤  std::vector<PurchaseRecord> purchases_;
};
```

c6/4/Portfolio.cpp

```
➤ const date Portfolio::FIXED_PURCHASE_DATE(date(2014, Jan, 1));

void Portfolio::Purchase(const string& symbol, unsigned int shareCount) {
   if (0 == shareCount) throw InvalidPurchaseException();
   holdings_[symbol] = shareCount + ShareCount(symbol);
➤  purchases_.push_back(PurchaseRecord(shareCount, FIXED_PURCHASE_DATE));
}

vector<PurchaseRecord> Portfolio::Purchases(const string& symbol) const {
   return purchases_;
}
```

代码很简单，但也花了几分钟才完成，每一点改动都有可能产生错误。在继续之前，获得对敲入代码的正面反馈是一件很好的事情。

我们定义了一个常量FIXED_PURCHASE_DATE，以便取得快速的、可以展示的进步。我们知道这是假设的。让我们去掉这个临时但有用的假设吧！

c6/5/PortfolioTest.cpp

```
TEST_F(APortfolio, AnswersThePurchaseRecordForASinglePurchase) {
➤  date dateOfPurchase(2014, Mar, 17);
➤  portfolio_.Purchase(SAMSUNG, 5, dateOfPurchase);

   auto purchases = portfolio_.Purchases(SAMSUNG);

   auto purchase = purchases[0];
   ASSERT_THAT(purchase.ShareCount, Eq(5u));
   ASSERT_THAT(purchase.Date, Eq(dateOfPurchase));
}
```

我们不调用Purchase()成员函数的其他测试，而是采取小步伐，使用一个默认的日期参数。

c6/5/Portfolio.h

```
void Purchase(
      const std::string& symbol,

      unsigned int shareCount,

      const boost::gregorian::date& transactionDate=
➤        Portfolio::FIXED_PURCHASE_DATE);
```

c6/5/Portfolio.cpp

```
➤ void Portfolio::Purchase(
➤       const string& symbol, unsigned int shareCount, const date& transactionDate) {
     if (0 == shareCount) throw InvalidPurchaseException();

     holdings_[symbol] = shareCount + ShareCount(symbol);
➤    purchases_.push_back(PurchaseRecord(shareCount, transactionDate));
  }
```

因为使用一个固定的日期不是有效的长期需求（尽管当前的默认时间可能是），所以我们现在想去除Purchase()的默认时间。不幸的是，我们还有许多测试调用了没有传入日期的Purchase()。

一种解决方案是，给所有测试中受影响的调用加一个日期参数。这似乎单调无味。同时，这也会违反测试抽象原则（参见7.4节），因为这些测试不关心购买日期。

需要一次性改变很多测试，且不改变其行为的事实告诉我们，这些测试蕴含着必须去除的重复。提供一个fixture辅助方法，由它来处理对Purchase()的调用并提供一个默认日期，这样测试就不用提供了，这个方案怎么样？

c6/6/PortfolioTest.cpp

```
  class APortfolio: public Test {
  public:
     static const string IBM;
     static const string SAMSUNG;
     Portfolio portfolio_;
     static const date ArbitraryDate;

➤    void Purchase(
➤          const string& symbol,
➤          unsigned int shareCount,
➤          const date& transactionDate=APortfolio::ArbitraryDate) {
➤       portfolio_.Purchase(symbol, shareCount, transactionDate);
➤    }
  };

  TEST_F(APortfolio, ReducesShareCountOfSymbolOnSell)  {
➤    Purchase(SAMSUNG, 30);
```

6

```
    portfolio_.Sell(SAMSUNG, 13);

    ASSERT_THAT(portfolio_.ShareCount(SAMSUNG), Eq(30u - 13));
}

TEST_F(APortfolio, AnswersThePurchaseRecordForASinglePurchase) {
    date dateOfPurchase(2014, Mar, 17);
➤   Purchase(SAMSUNG, 5, dateOfPurchase);

    auto purchases = portfolio_.Purchases(SAMSUNG);

    auto purchase = purchases[0];
    ASSERT_THAT(purchase.ShareCount, Eq(5u));
    ASSERT_THAT(purchase.Date, Eq(dateOfPurchase));
}
```

一个可能引起争议的点是，辅助函数`Purchase()`从测试中移除了一些信息——具体而言，它委派给了`portfolio_`实例。第一次阅读测试的人必须查看辅助函数以便了解其作用。但这是一个简单函数，没有隐藏对阅读者来说难以记忆的关键信息。

根据经验，不要隐藏事实（参见4.2节）。当需要让买入操作成为设置测试的一部分时，我们可以使用辅助函数。但在专门测试`Purchase()`行为时，我们应该直接调用它。因此，ReducesShareCountOfSymbolOnSell可以使用辅助函数，因为买入操作是设置测试的一部分。因为AnswersShareCountForPurchasedSymbol是验证买入行为的，所以其中保留了对`Purchase()`的直接调用。

c6/7/PortfolioTest.cpp

```
TEST_F(APortfolio, AnswersShareCountForPurchasedSymbol) {
➤   portfolio_.Purchase(IBM, 2);
    ASSERT_THAT(portfolio_.ShareCount(IBM), Eq(2u));
}

TEST_F(APortfolio, ReducesShareCountOfSymbolOnSell)  {
➤   Purchase(SAMSUNG, 30);

    portfolio_.Sell(SAMSUNG, 13);
    ASSERT_THAT(portfolio_.ShareCount(SAMSUNG), Eq(30u - 13));
}
```

就个人而言，我不喜欢这种方式在测试中导致的不一致性。我可以接受用辅助函数运行**一切**，只要它是一个简单的单行委托，就像本例一样。如果这让你感到困扰，另一种方案是，在每个测试中包含一个日期参数，并使用一个名为`ArbitraryPurchaseDate`的常量。

我们一直以非常小的步骤采取增量方法。这耗费时间吗？当然！经常需要引入少量的代码，然后再移除——有一点点浪费。

作为回报，我们获得了更有价值的能力，即能够在创建设计良好、正确的代码道路上持续迈

进。不用为处理新的、从未考虑的功能而患得患失——用TDD以一系列小的改动来开发它们。代码越是整洁，越容易改动。

6.2.5 更多的重复

通过一系列的测试和产品代码清理，针对买入记录的测试变得简短且干净。

c6/8/PortfolioTest.cpp

```
TEST_F(APortfolio, AnswersThePurchaseRecordForASinglePurchase) {
   Purchase(SAMSUNG, 5, ArbitraryDate);

   auto purchases = portfolio_.Purchases(SAMSUNG);
   ASSERT_PURCHASE(purchases[0], 5, ArbitraryDate);
}

TEST_F(APortfolio, IncludesSalesInPurchaseRecords) {
   Purchase(SAMSUNG, 10);
   Sell(SAMSUNG, 5, ArbitraryDate);

   auto sales = portfolio_.Purchases(SAMSUNG);
   ASSERT_PURCHASE(sales[1], -5, ArbitraryDate);
}
```

为了支持负数的买入记录，我们将`ShareCount`改为有符号整数。

c6/8/Portfolio.h

```
struct PurchaseRecord {
➤  PurchaseRecord(int shareCount, const boost::gregorian::date& date)
      : ShareCount(shareCount)
      , Date(date) {}
➤  int ShareCount;
   boost::gregorian::date Date;
};
```

c6/8/PortfolioTest.cpp

```
void ASSERT_PURCHASE(
➤     PurchaseRecord& purchase, int shareCount, const date& date) {
   ASSERT_THAT(purchase.ShareCount, Eq(shareCount));
   ASSERT_THAT(purchase.Date, Eq(date));
}
```

在产品代码中，用作交易的`Purchase()`和`Sell()`函数中的三行代码看起来有点密。

c6/8/Portfolio.cpp

```
void Portfolio::Purchase(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   if (0 == shareCount) throw InvalidPurchaseException();
   holdings_[symbol] = shareCount + ShareCount(symbol);
   purchases_.push_back(PurchaseRecord(shareCount, transactionDate));
```

```
}

void Portfolio::Sell(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   if (shareCount > ShareCount(symbol)) throw InvalidSellException();
   holdings_[symbol] = ShareCount(symbol) - shareCount;
   purchases_.push_back(PurchaseRecord(-shareCount, transactionDate));
}
```

再进一步来说，它们本质上很相似。我们还没有编写合适的逻辑将买入记录和正确的股票名称对应（或许可以用哈希表），也不想重复编码。让我们先看看可以去除哪些重复。

`Purchase()`和`Sell()`函数的这三行代码差别不大。让我们逐条审阅每行代码，来看看怎样让它们相似。两个函数中的第一行都是保护语句，用于执行一个约束：卖出的数量不能比持有的多，不可以买入0股。但是卖出是不是也应该有相同的约束，即不能卖出0股？客户认为这是可以的。

还有一个小问题是，在`Sell()`函数中使用InvalidPurchaseException异常类型的名称是不太合适的。我们将它具体化，这样两个函数都可以使用——ShareCountCannotBeZeroException。

c6/9/PortfolioTest.cpp

```
TEST_F(APortfolio, ThrowsOnPurchaseOfZeroShares) {
   ASSERT_THROW(Purchase(IBM, 0), ShareCountCannotBeZeroException);
}
// ...
TEST_F(APortfolio, ThrowsOnSellOfZeroShares) {
   ASSERT_THROW(Sell(IBM, 0), ShareCountCannotBeZeroException);
}
```

这样一来，两个交易函数的保护语句就一样了。

c6/9/Portfolio.cpp

```
void Portfolio::Purchase(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
➤  if (0 == shareCount) throw ShareCountCannotBeZeroException();
   holdings_[symbol] = shareCount + ShareCount(symbol);
   purchases_.push_back(PurchaseRecord(shareCount, transactionDate));
}

void Portfolio::Sell(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   if (shareCount > ShareCount(symbol)) throw InvalidSellException();
➤  if (0 == shareCount) throw ShareCountCannotBeZeroException();
   holdings_[symbol] = ShareCount(symbol) - shareCount;
   purchases_.push_back(PurchaseRecord(-shareCount, transactionDate));
}
```

继续看两个函数中的下一行代码，通过加或减已持有的股票数，可以更新相应股票名称的持有量。但是减和加上对应负数的效果是一样的。

我们引入一个名为shareChange的有符号整数变量来做到这一点。注意，也可以在最后一行代码（添加买入记录的地方）中使用它。

c6/10/Portfolio.cpp

```
void Portfolio::Sell(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   if (shareCount > ShareCount(symbol)) throw InvalidSellException();
   if (0 == shareCount) throw ShareCountCannotBeZeroException();
➤  int shareChange = -shareCount;
➤  holdings_[symbol] = ShareCount(symbol) + shareChange;
➤  purchases_.push_back(PurchaseRecord(shareChange, transactionDate));
}
```

现在再回头看看Purchase()函数，并尝试让它看起来更像Sell()。

c6/11/Portfolio.cpp

```
void Portfolio::Purchase(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   if (0 == shareCount) throw ShareCountCannotBeZeroException();
➤  int shareChange = shareCount;
➤  holdings_[symbol] = ShareCount(symbol) + shareChange;
➤  purchases_.push_back(PurchaseRecord(shareChange, transactionDate));
}
```

现在，每个函数的末尾两行彼此重复，保护语句也是如此。让我们将shareChange变量的初始化提到保护语句前。虽然将代码行移上移下是一件风险极高的事，但现有的测试会确保此改动的安全性。

6

最终得到的每个函数，末尾三行相同。同时，我们将保护语句中使用的shareCount更名为shareChange，这样，待提取出来的三行代码使用相同的变量。

c6/12/Portfolio.cpp

```
void Portfolio::Purchase(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   int shareChange = shareCount;
➤  if (0 == shareChange) throw ShareCountCannotBeZeroException();
➤  holdings_[symbol] = ShareCount(symbol) + shareChange;
➤  purchases_.push_back(PurchaseRecord(shareChange, transactionDate));
}

void Portfolio::Sell(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   if (shareCount > ShareCount(symbol)) throw InvalidSellException();
   int shareChange = -shareCount;
➤  if (0 == shareChange) throw ShareCountCannotBeZeroException();
➤  holdings_[symbol] = ShareCount(symbol) + shareChange;
➤  purchases_.push_back(PurchaseRecord(shareChange, transactionDate));
}
```

最后，可以提取了。

c6/13/Portfolio.cpp

```
void Portfolio::Purchase(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   Transact(symbol, shareCount, transactionDate);
}

void Portfolio::Sell(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
   if (shareCount > ShareCount(symbol)) throw InvalidSellException();
   Transact(symbol, -shareCount, transactionDate);
}
void Portfolio::Transact(
      const string& symbol, int shareChange, const date& transactionDate) {
   if (0 == shareChange) throw ShareCountCannotBeZeroException();
   holdings_[symbol] = ShareCount(symbol) + shareChange;
   purchases_.push_back(PurchaseRecord(shareChange, transactionDate));
}
```

另外一个和表达力相关的事情是，异常类型InvalidSellException的名称不是很好。我们将其改成InsufficientSharesException。

c6/14/PortfolioTest.cpp

```
TEST_F(APortfolio, ThrowsWhenSellingMoreSharesThanPurchased) {
➤  ASSERT_THROW(Sell(SAMSUNG, 1), InsufficientSharesException);
}
```

c6/14/Portfolio.cpp

```
void Portfolio::Sell(
      const string& symbol, unsigned int shareCount, const date& transactionDate) {
➤  if (shareCount > ShareCount(symbol)) throw InsufficientSharesException();
   Transact(symbol, -shareCount, transactionDate);
}
```

除了秉承两个简单的设计规则外，还有什么可以做得吗？我们似乎已经清除了所有的重复。那可读性呢？除了委派工作，`Purchase()`什么都没做，因此很清晰。`Sell()`只是简单地加了一个约束并将股票数取反，所以也比较清晰。但`Transact()`完全没有我们想要的即时性①。

6.2.6　小方法的好处

阅读`Transact()`时，必须放慢速度，搞清楚每行代码到底要完成什么。第一行代码会在股票数为0时抛出异常。第二行代码获取相应股票的数量，加上股数改动后，再将结果赋值给哈希表中对应的项。第三行代码创建了一个买入记录，并将其加入到一个总的买入列表中。

`Transact()`函数只包含三行简单的代码。但如果需要仔细地阅读每行代码的话，那么整个

① 众所周知，股票交易讲究即时性。作者的言下之意是指`Transact()`实现感觉上做了许多事。这也为下一小节埋下了伏笔。——译者注

系统阅读起来会更困难。这不具表达力。我们可以对其进行修改。

c6/15/Portfolio.cpp

```
void Portfolio::Transact(
      const string& symbol, int shareChange, const date& transactionDate) {
   ThrowIfShareCountIsZero(shareChange);
   UpdateShareCount(symbol, shareChange);
   AddPurchaseRecord(shareChange, transactionDate);
}
void Portfolio::ThrowIfShareCountIsZero(int shareChange) const {
   if (0 == shareChange) throw ShareCountCannotBeZeroException();
}

void Portfolio::UpdateShareCount(const string& symbol, int shareChange) {
   holdings_[symbol] = ShareCount(symbol) + shareChange;
}

void Portfolio::AddPurchaseRecord(int shareChange, const date& date) {
   purchases_.push_back(PurchaseRecord(shareChange, date));
}
```

我有着超强的额外感知力。在2013年年初创作本章时，我仿佛就能看到许多读者的表情。我能感受到你们的惊愕，也能理解。

下面可能是你会提出不这么做的一些理由。

- 这是额外功。创建新函数很费力。
- 为只在一个地方用到的单行代码创建一个函数似乎很荒唐。
- 额外的函数调用会加重性能开销。
- 很难在所有代码中遵循完整的控制流。
- 你会得到成千上万个小方法，且每个方法都有着巨长的名称。

下面是你应该考虑这样写代码的一些原因。

这遵守了表达力这一简单设计原则。代码不再需要解释性注释。可以快速理解由一行或多行语句构成的函数，马上看出小函数出现的问题。

相比之下，大多数系统有很多代码过长且密集的函数，让人难以理解。这些函数很容易藏匿问题。

- 这遵守了内聚和单一责任的设计原则。同一函数中的所有代码处于同一层次的抽象。修改每个函数的原因只有一个。
- 这为以后的设计改动铺平了道路。我们仍然需要将买入记录和对应的股票名称联系起来，但现在可以在一处完成改动，而非两处。如果需要特别处理`AddPurchaseRecord()`，现在就可以做了。如果需要创建更复杂的买入记录子系统，我们可以快速地将已有的逻辑移到一个新类。如果需要支持取消和重做，或围绕买卖做些其他事情，我们已经准备好

了以Transcation为基类的命令模式。

- 忽略具体的实现细节会更容易理解代码控制流。Transact()只是一个声明策略。其他辅助函数，如ThrowIfShareCountIsZero()、UpdateShareCount()和AddPurchaseRecord()，则是你基本不需要知道的实现细节。回忆一下接口和实现或抽象和具体的分离理念。
- 提取方法导致的性能开销几乎不会造成问题。参见10.2节。
- 小方法是真正重用的开始。随着越来越多的相似函数被提取出来，识别出重复的概念和结构会变得更加容易。你不会得到一大堆不可控的小函数，相反，你会大大地缩减产品代码的数量。

差不多就是这些了。如果还没准备好接受这种风格的巨变，那么随着更加深入地实践TDD，你至少应当为之努力。多使用小的函数，看看会发生什么。

6.2.7 完成功能

我们还没有完成。目前的投资管理器能够返回一个买入列表，但只能返回一支股票的列表。下一个测试要求投资管理器能返回多支已购股票的买入记录。

c6/16/PortfolioTest.cpp

```
bool operator==(const PurchaseRecord& lhs, const PurchaseRecord& rhs) {
   return lhs.ShareCount == rhs.ShareCount && lhs.Date == rhs.Date;
}

TEST_F(APortfolio, SeparatesPurchaseRecordsBySymbol) {
   Purchase(SAMSUNG, 5, ArbitraryDate);
   Purchase(IBM, 1, ArbitraryDate);

   auto sales = portfolio_.Purchases(SAMSUNG);
   ASSERT_THAT(sales, ElementsAre(PurchaseRecord(5, ArbitraryDate)));
}
```

Google Mock提供了ElementsAre()匹配器，用来验证指定的元素是否在集合中。这一比较需要比较两个PurchaseRecord对象的能力，所以我们添加了operator==()的正确实现。（也可以将opeator()==实现为PurchaseRecord的成员函数，但目前我们仅在测试中才需要它。）测试一开始失败了，因为名为purchase_的向量容器只包含两个买入记录——一个是Samsung，另一个是IBM。

为了让测试通过，我们先在Portfolio.h中定义成员变量purchaseRecord_，这是一个无序映射容器，为每支股票保存了一个包含了此股票买入记录的向量容器。同时，我们修改AddPurchaseRecord()函数，使其接受一支股票作为参数。

c6/16/Portfolio.h

```
class Portfolio {
```

```
public:
   bool IsEmpty() const;
   void Purchase(
         const std::string& symbol,
         unsigned int shareCount,
         const boost::gregorian::date& transactionDate);
   void Sell(const std::string& symbol,
         unsigned int shareCount,
         const boost::gregorian::date& transactionDate);
   unsigned int ShareCount(const std::string& symbol) const;
   std::vector<PurchaseRecord> Purchases(const std::string& symbol) const;

private:
   void Transact(const std::string& symbol,
         int shareChange,
         const boost::gregorian::date&);
   void UpdateShareCount(const std::string& symbol, int shareChange);
➤  void AddPurchaseRecord(
➤        const std::string& symbol,
➤        int shareCount,
➤        const boost::gregorian::date&);
   void ThrowIfShareCountIsZero(int shareChange) const;

   std::unordered_map<std::string, unsigned int> holdings_;
   std::vector<PurchaseRecord> purchases_;
➤  std::unordered_map<std::string, std::vector<PurchaseRecord>> purchaseRecords_;
};
```


相应地，我们也更新了Transact()的实现，将股票名传给AddPurchaseRecord()。在AddPurchaseRecord()中，通过加入新的代码，将PurchaseRecord添加进purchaseRecords_中（如果需要的话，先添加一个空的向量容器）。保持已有的逻辑不变——在整理旧代码前，先让新的代码如期工作。

c6/16/Portfolio.cpp

```
void Portfolio::Transact(
      const string& symbol, int shareChange, const date& transactionDate) {
   ThrowIfShareCountIsZero(shareChange);
   UpdateShareCount(symbol, shareChange);
➤  AddPurchaseRecord(symbol, shareChange, transactionDate);
}

void Portfolio::AddPurchaseRecord(
      const string& symbol, int shareChange, const date& date) {
   purchases_.push_back(PurchaseRecord(shareChange, date));
➤  auto it = purchaseRecords_.find(symbol);
➤  if (it == purchaseRecords_.end())
➤     purchaseRecords_[symbol] = vector<PurchaseRecord>();
➤  purchaseRecords_[symbol].push_back(PurchaseRecord(shareChange, date));
}

unsigned int Portfolio::ShareCount(const string& symbol) const {
   auto it = holdings_.find(symbol);
```

```
    if (it == holdings_.end()) return 0;
    return it->second;
}
vector<PurchaseRecord> Portfolio::Purchases(const string& symbol) const {
➤ //   返回purchases_;
➤   return purchaseRecords_.find(symbol)->second;
}
```

在Purchases()函数中，根据股票名返回相应的买入记录向量容器。我们只写适量的代码，不用担心能不能找到股票。相比于担心，我们在测试列表中加入一个条目（“处理在买入记录中找不到特定股票的情况”）。

一旦所有测试通过，就可以通过移除对purchase_的引用来整理代码。编写刚刚加入测试列表的测试。我们先进一步整理一下代码。从表达力角度来看，AddPurchaseRecord()显得有点攒簇。从重复代码角度来看，ShareCount()和Purchases()中都包含了在映射容器中查找元素的代码。我们先处理这两个问题。

c6/17/PortfolioTest.cpp

```
TEST_F(APortfolio, AnswersEmptyPurchaseRecordVectorWhenSymbolNotFound) {
   ASSERT_THAT(portfolio_.Purchases(SAMSUNG), Eq(vector<PurchaseRecord>()));
}
```

c6/17/Portfolio.h

```
template<typename T>
T Find(std::unordered_map<std::string, T> map, const std::string& key) const {
   auto it = map.find(key);
   return it == map.end() ? T{} : it->second;
}
```

c6/17/Portfolio.cpp

```
#include "Portfolio.h"
#include "PurchaseRecord.h"
using namespace std;
using namespace boost::gregorian;

bool Portfolio::IsEmpty() const {
   return 0 == holdings_.size();
}

void Portfolio::Purchase(
      const string& symbol,
      unsigned int shareCount,
      const date& transactionDate) {
   Transact(symbol, shareCount, transactionDate);
}

void Portfolio::Sell(
      const string& symbol,
      unsigned int shareCount,
      const date& transactionDate) {
```

```
   if (shareCount > ShareCount(symbol)) throw InvalidSellException();
   Transact(symbol, -shareCount, transactionDate);
}

void Portfolio::Transact(
      const string& symbol, int shareChange, const date& transactionDate) {
   ThrowIfShareCountIsZero(shareChange);
   UpdateShareCount(symbol, shareChange);
   AddPurchaseRecord(symbol, shareChange, transactionDate);
}

void Portfolio::ThrowIfShareCountIsZero(int shareChange) const {
   if (0 == shareChange) throw ShareCountCannotBeZeroException();
}

void Portfolio::UpdateShareCount(const string& symbol, int shareChange) {
   holdings_[symbol] = ShareCount(symbol) + shareChange;
}

void Portfolio::AddPurchaseRecord(
      const string& symbol, int shareChange, const date& date) {
   if (!ContainsSymbol(symbol))
      InitializePurchaseRecords(symbol);
   Add(symbol, {shareChange, date});
}

void Portfolio::InitializePurchaseRecords(const string& symbol) {
   purchaseRecords_[symbol] = vector<PurchaseRecord>();
}
void Portfolio::Add(const string& symbol, PurchaseRecord&& record) {
   purchaseRecords_[symbol].push_back(record);
}

bool Portfolio::ContainsSymbol(const string& symbol) const {
   return purchaseRecords_.find(symbol) != purchaseRecords_.end();
}

unsigned int Portfolio::ShareCount(const string& symbol) const {
   return Find<unsigned int>(holdings_, symbol);
}

vector<PurchaseRecord> Portfolio::Purchases(const string& symbol) const {
   return Find<vector<PurchaseRecord>>(purchaseRecords_, symbol);
}
```

好吧，我们整理了不只“一点点”，难道不是吗？我们再一次做了大量的小函数重构。现在`AddPurchaseRecord()`声明了高层次的策略，其中的三个函数代表了策略中封装了实现细节的每个步骤。过度编码？或许吧！优势呢？可以立刻理解代码，这是肯定的。同时，实现细节的分离意味着，如果想要使用不同的数据结构，需要改动的地方将会一目了然且相对独立，因此降低了风险。此外，由于对合适的成员函数使用了`const`，我们也可以清楚地知道修改了Portfolio状态的每个步骤。

6

最后，我们也做好了迈向更好设计的准备。在下一节中，买入记录的集合最终自成一体，成为一个独立的类。目前，对所有基于此集合操作的整理为切换到新的设计提供了方法。

要说明的是，我们并没有提前思考设计。相反，我们先得到了一个可以运行的代码，然后再优化现有方案的设计。这样做的副作用是，以后的改动会更加容易。

6.2.8 增量设计让事情变得简单

Portfolio中有两个很相似的集合：holdings_和purchaseRecords_，前者将股票和总股数相联系，后者将股票和购买记录相联系。可以去除holdings_，转而按照需要来计算一个特定股票的股数。

保持两个集合是一种性能优化。这样导致代码稍显复杂，我们需要保证这两个集合总是保持一致。这最终还是由你决定的。如果你认为性能优先，那么可以保持代码不变。因为目前不需要考虑性能，所以先找出共同的代码。

第一步是改变ShareCount()，使其能够动态地计算出特定股票的股数。

c6/18/Portfolio.cpp

```
unsigned int Portfolio::ShareCount(const string& symbol) const {
   auto records = Find<vector<PurchaseRecord>>(purchaseRecords_, symbol);
   return accumulate(records.begin(), records.end(), 0,
      [] (int total, PurchaseRecord record) {
            return total + record.ShareCount; });
}
```

我们不需要在Transact()中调用UpdateShareCount()了。可以安全地删除UpdateShareCount()了！然后修改IsEmpty()，使其指向purchaseRecords_而非holdings_，这样我们终于可以删除holdings_了。

c6/18/Portfolio.cpp

```
bool Portfolio::IsEmpty() const {
➤   return 0 == purchaseRecords_.size();
}

void Portfolio::Transact(
      const string& symbol, int shareChange, const date& transactionDate) {
   ThrowIfShareCountIsZero(shareChange);
   AddPurchaseRecord(symbol, shareChange, transactionDate);
}
```

整个过程十分简单。

最后要做的是，将所有和买入记录相关的代码抽出来，单独创建一个类。为什么呢？Portfolio类违反了单一责任原则。修改Portfolio类的主要原因应该和操作股票的方式有关。但还有一个需要修改的原因——针对买入记录的特定实现细节。

那又怎样呢？我们只是改动了设计来简化代码，但是这个改动可能也象征着不可接受的性能退化。将买入记录代码独立出来遵循了单一责任原则的设计，让我们可以更容易地找到需要性能优化的点。提取代码可以降低不小心破坏Portfolio功能的概率。

我们可以再一次增量地改动代码、添加一些新代码、运行测试以确保所有的功能依然有效。首先，引入一个新的成员变量，从而将股票和持有量联系起来。可以将之命名为`holdings_`（嘿，听着很耳熟啊！）。

c6/19/Portfoilo.h

```
➤ std::unordered_map<std::string, Holding> holdings_;
```

然后，逐步加入支持来更新`holdings_`，先从`initializePurchaseRecords()`开始。

6/19/Portfolio.cpp

```
  void Portfolio::InitializePurchaseRecords(const string& symbol) {
     purchaseRecords_[symbol] = vector<PurchaseRecord>();
➤    holdings_[symbol] = Holding();
  }
```

在`Add()`函数中，将功能委托给Holding类中的一个同名函数。从Portfolio类中复制代码到Holding类，并适当简化。

c6/19/Portfolio.cpp

```
  void Portfolio::Add(const string& symbol, PurchaseRecord&& record) {
     purchaseRecords_[symbol].push_back(record);
➤    holdings_[symbol].Add(record);
  }
```

c6/19/Holding.h

```
void Add(PurchaseRecord& record) {
   purchaseRecords_.push_back(record);
}
std::vector<PurchaseRecord> purchaseRecords_;
```

现在轮到`Purchases()`函数了。用一个更简单的版本替代现有的代码（先注释掉——不要担心，这些被注释的代码不会一直这样的），将功能也委托给Holding类中的一个新函数。

c6/19/Portfolio.cpp

```
vector<PurchaseRecord> Portfolio::Purchases(const string& symbol) const {
//   return Find<vector<PurchaseRecord>>(purchaseRecords_, symbol);
   return Find<Holding>(holdings_, symbol).Purchases();
}
```

c6/19/Holding.h

```
std::vector<PurchaseRecord> Purchases() const {
   return purchaseRecords_;
}
```

就像之前的purchaseRecords_，对holdings_做相同的操作可以更新ContainsSymbol()。

c6/19/Portfolio.cpp

```
bool Portfolio::ContainsSymbol(const string& symbol) const {
//   return purchaseRecords_.find(symbol) != purchaseRecords_.end();
   return holdings_.find(symbol) != holdings_.end();
}
```

修改ShareCount()是另一种委派。

c6/19/Portfolio.cpp

```
unsigned int Portfolio::ShareCount(const string& symbol) const {
//   auto records = Find<vector<PurchaseRecord>>(purchaseRecords_, symbol);
//   return accumulate(records.begin(), records.end(), 0,
//      [] (int total, PurchaseRecord record) {
//          return total + record.ShareCount; });
   return Find<Holding>(holdings_, symbol).ShareCount();
}
```

c6/19/Holding.h

```
unsigned int ShareCount() const {
   return accumulate(purchaseRecords_.begin(), purchaseRecords_.end(), 0,
      [] (int total, PurchaseRecord record) {
          return total + record.ShareCount; });
}
```

最后，我们尝试移除成员变量purchaseRecords_。编译器会指出哪些代码仍在引用此变量。删掉这些引用并确保测试依然通过。（确实通过了！）

c6/19/Portfolio.h

```
// 不再需要了！
std::unordered_map<std::string, std::vector<PurchaseRecord>> purchaseRecords_;
```

差不多大功告成了。最后的工作是确保为Holding类创建了测试。代码已经在Portfolio的上下文中测试过了，为什么还要添加测试呢？原因是我们也想保留测试提供的文档价值。如果想使用Holding类，开发者可以通过阅读测试（记录了它的行为）来了解怎么使用。

当以此方式提取新类时，有时可以将测试直接转移（例如，从PortfolioTest到HoldingTest）。通常而言，测试也会变得简单，这时可能要考虑重新为它们起个名字。（可以看看下载的源代码中的最后测试）。

最后的结果是Holding类中的代码极其简单，每个函数只有一行，很容易理解。Portfolio类中的代码也相当简单，函数也都只有一两行，也很容易理解。

贯穿在整个过程中的另一件美妙事情是，我们可以对设计一步步地做出大的改动，不需要有太多顾虑。坦率地说，在为本书编写这个示例时，我至少犯了一个不易察觉的错误，好在测试立刻让我明白错在哪了。

至于性能，恢复到最初的性能还是简单直接的。如果需要，我们可以将股票总数缓存在Holding类的一个成员变量中，每次调用`Add()`时就将数量累加到这个变量，用`ShareCount()`直接返回这个变量的值。

6.3 预先设计在哪

如果是在20世纪90年代编写代码的话，你的团队必定会做大量的前期工作来完成设计模型。可能要研究用例来理解需求，接下来要创建类图、状态模型、顺序图、协作图、组件模型，等等。

如今，你可能很少会创建此类模型。敏捷方法的关注点使开发团队抛弃了详细的设计模型。“敏捷方法认为我们不需要设计”，至少我听说过这句话。

那么敏捷中所说的设计是什么呢？敏捷原则（敏捷宣言的一部分，参见http://agilemanifesto.org/principles.html）声明，软件必须对客户有用，能够在任何时候适应变化的需求，同时必须具备可以工作、拥有好的设计、不包含没必要的复杂度等特性。此原则中没有对“好”的定义，也没提到什么时候设计。

敏捷的挑战在于要能应对变化。譬如，一个新的功能需求出现了，但这个功能你从来没想到过，现在你必须想出怎样将此功能加入到并没有为适应此功能而设计的系统中。

假设你花费了大量的时间来决定整个新系统必须要做什么。得出了相应设计模型的完美集合后，开始构建系统。因为没有花时间做TDD，所以你的团队进展很快（至少他们是这么认为的）。分析、设计和实现的每个步骤各花两个月，这意味着六个月之后，你就可以发布系统了。

此后冒出的任何新需求没有体现在你的全面预先设计中。有时这没什么问题，但是大部分时候却不是如此，这时系统开发的速度开始减慢。你能够简单地处理一些新的功能需求，但是很多其他改动很难实现。有些改动需要令人厌烦的特殊处理，因为系统设计并没有考虑它们。许多新功能需要在系统范围内做大量改动。分析系统、得出怎样进行这些改动所花费的时间也在增加。

随着系统中的依赖结构退化，开发的速度会持续变慢。当新的改动引入难于修复且看似不相干的缺陷时，团队的进度会进一步变慢。此时你会希望用快速单元测试的方式来更好地驾驭系统。

然而，即使知道系统必须要做什么，勾勒出最初的设计还是有价值的。但没必要在初始设计的细节方面做到巨细无遗。要关注高层结构：哪些是系统中的关键类和依赖关系？子系统之间以及对外的接口有哪些？核心消息流又有哪些？你可以只花费两个月的零头来起草一份良好的高层结构和初始设计，不这样做则需要多花费两个月时间。

TDD是处理设计的一种方法，具有持续性。好比把初始设计期间节省下来的时间分摊到整个产品周期中。研究（参见11.2节）表明，TDD在项目伊始时会耗费多一点的开发精力，但产出的代码是高质量的。这些研究没有讨论最小化预先设计节省的时间。

当然，可以在预先设计期投入大量精力，但要接受所得出的模型几乎总是错的。一旦开始编码，许多变化就会出现：客户改变了主意、市场发生了变化、你学到了更多东西并发现了更好的设计方法，或者某人意识到漏掉了一些需求或有些需求错了。

预先设计是一个很好的起始路线图。围绕此设计的讨论有助于发现软件中必须要做的东西，以及最初该怎样设计软件。但是，打造系统所需要的大量细节会发生变化。例如，类图是一个需要创建的好东西，但不要过度执着于底层细节：私有还是公开，属性细节，聚合还是组合，等等。这些东西来源于测试驱动的过程。相反，应致力于类名、依赖关系，或一些关键的公共行为。

TDD允许你基于当前的业务需求，保持一个可能的最简设计。如果一直保持设计尽量简洁，那么就可以最大可能地引入新的、从未被考虑过的功能。相反，如果任由系统退化（并有大量的重复代码和晦涩难懂的代码），未来有任何新的需求时，你将痛苦万分。

6.3.1　哪里才会讨论真正的设计呢

我们在前文中就讨论了真正的设计。可以重读一下Portfolio示例。诚然，这只是应用程序的一个简单部分，但是其中所用的理念却适用于超大型系统。

“不，我的意思是，哪里讨论了耦合、内聚、单一责任原则、SOLID设计原则、依赖结构、代码坏味、迪米特法则[①]、模式、封装、组合还是继承，等等？”问得好。首先，本书不是一本讨论经典面向对象设计概念的图书。本书讨论的是测试驱动开发，本章的目标是向你展示怎样增量地处理持续进化的设计。

其次，你真的需要知道所有的经典设计理念。同时也要一直寻求更多关于设计的知识。只要能比较容易地增量改动就还好。如果目前还不知道这些设计理念，不要担心。简单的设计理念[②]将让你抵达设计殿堂。但要继续深入学习。

当处于TDD的重构阶段，你要尽可能地了解与优秀设计构成相关的所有知识。同时，也要尽可能地了解团队的想法。你是在一个共享的代码库中工作，需要与团队就哪些可接受、哪些与设计无关等方面达成共识。

大多数时候，经典设计理念和简单设计原则相一致。举个例子，设计模式主要与解决方案的表达力相关。像模板方法这样的模式主要用于消除重复。

① 迪米特法则是Ian Holland在美国东北大学做Demeter项目时提出的，此法则命名也是来源于此项目。此规则的目的是通过简单的方法减少系统中对象间的行为耦合，每个对象尽量减少与其他对象的直接交互行为。此规则后来被许多工业系统所采用，包括NASA的火星探路者飞行器系统。——译者注

② 简单的设计理念在Unix程序设计文化中被奉为圭臬，即常说的K.I.S.S（Keep it simple, stupid）原则。和模式一样，此原则来自于计算机外的工程领域。由美国航空系统工程师Kelly Jonhson首次提出。——译者注

6.3.2 简单设计原则和经典设计理念会在哪起冲突

在一些情况下，你应该以现代、增量的设计理念取代经典（又称为旧的）的设计理念。下面列出了老派设计和TDD中的增量设计相冲突的一些地方。

- **访问性**：仍然应当尽量保持成员私有化。这会使得有些改动变得更容易些（例如，安全地重命名一个公有成员函数所需要的工作量比重命名一个私有函数要多）。虽然不太可能，但暴露不必要的成员会使得系统受到恶意或糊涂客户的破坏。

  但是，如果你需要放松访问控制来让测试验证一些功能是否可以如期工作，那大多数时间就不要为此担心了。如果每样东西都被测试，那么测试就会保护系统免受糊涂客户的破坏。知道系统可以如期工作远比对未来滥用的杞人忧天更让人向往。如果你依然感到担忧，也有比较聪明且安全的方法。但记住，聪明和愚蠢往往只是一念之差。

  在测试中，绝对要避免不必要的设计，例如，用私有还是公有控制。没有人会调用你的测试，测试中的访问指示符只会影响可读性。

- **及时性**：老派的设计希望你能尝试获得尽可能完美的设计。在简单设计中，这是不正确的。实际上，越是绞尽脑汁想出应对未来每种可能功能的设计，就越会付出更多时间，同时，当功能需求真的出现时，你依然需要做大量的修改。最好的方式是，学习怎样通过简单、增量的设计来持续地应对变化。

6.4 阻碍重构的因素

秉承将简单、增量的设计理念作为主要的设计驱动力的原则，重构环节是完成很多真正的工作的地方。任何阻碍你方便地重构，甚至打消重构想法的因素都是不好的，可以说是糟糕的。一旦停止增量地重构，系统就会迅速地退化。

当心以下阻碍重构的因素。

- **测试不足**：使用TDD的话，构建进系统的每一小块逻辑都有对应的快速测试。这些测试可以给予你充足的信心来构建更好的代码。相反，如果你只有少量的快速单元测试，随之而来的是更低的测试覆盖率，重构的热情和重构的能力会大大缩减。开发代码的方法也会趋于保守："如果没有故障，那就不用改动！"例如，你明知道将相同的代码改为一个辅助函数是好事，但是你不做，因为这些改动会涉及别人的代码，而这些代码已经可以工作。
- **生存周期长的分支**：如果曾经合并过包含大量改动的其他分支中的代码，那么你应该知道大量的重构会使合并代码异常困难。在一个分支上工作的开发者或许会被要求最小化改动范围。这样做可能会使代码合并更简单些，但也会使代码库从此深受煎熬[①]。如果必

① 大量的小改动会使代码库的代码提交增加，虽然多点，但小的改动更符合TDD风格。——译者注

须在长时间内维持多个分支，那么你可以持续地从主线中合并代码。否则，避免生存周期长的分支。

- **与实现相关的测试**：在测试驱动开发中，类的行为是通过其公有接口来展现的。按定义来说，重构是在不改变其外在公有行为的前提下改变设计的。如果测试知道内部私有实现细节的话，那么在这些私有细节改变时，测试就有失败的可能。你要能够按需地改变底层代码结构，自由地提取或内联方法。

  大量的模拟或协作者存根会向测试暴露一些本该私有的细节。使用得当的话，使用测试替身不会导致问题。但如果随意使用，你可能会在重构时发现许多测试失败。这也是很多开发者不愿意重构的好借口。

- **大量的技术债务**：大量晦涩难懂的代码足以让许多开发者放弃重构。“我该从哪下手呢？”越是任由代码退化，越是难以对它做出改变。一定要确保始终好好利用重构环节。
- **缺乏知识**：任何你不知道的事情都能够且终将绊倒你。或许你对设计略知一二，在本章中也学习了简单设计原则，但还是再多学点吧！如果在设计方面没有夯实基础，你的重构就很可能不够充分。
- **着迷于提前的性能优化**：本书中所提倡的许多观点都是基于小的类和函数，它们会带来创建额外对象、调用额外函数的开销。许多开发者对此有点抵触，并满意于过长的函数和类。

  确保先创建一个整洁、可维护的设计。得到一个适合设计的性能数据来判断它是否有性能缺陷。仅对必须优化的代码进行优化。大多数优化会增加理解和维护代码的难度。

- **绩效考核**：如果公司崇尚绩效或将一大笔钱（以工资或奖金的方式）与特定的指标绑定，那么聪明的员工会竭尽所能地完成指标。如果指标是以一个数字度量的，那么精明的开发者会寻找达到这一数字的最优方法，而不管其所做所为是否是真正业务所需要的。

  例如，将缺陷密度值定义为每一千行代码的缺陷数目（kilo lines of code，KLOC）。（注意，你也可以将缺陷密度看作每个功能点[①]有多少个缺陷，但是计算功能点却不是那么简单的。因此大多数团队使用简单的度量，缺陷/KLOC。）如果你的经理过度强调缩减缺陷密度值，团队也会相应地作出反应。确保代码出现更少的缺陷比玩弄数值要难得多。最简单的方法就是增加代码行数。

  或许你相信大部分程序员不会那么狡诈。或许不会吧！但是当你要求他们将两个几乎一样的、包含千行的函数合成一个时，他们就会想到在KLOC上的损失，这样会增加缺陷密度值。想要说服他们相信消除重复代码的重要性，只能祝你好运了。

---

① 功能点是描述信息系统向用户提供了多少功能的度量单位。可以参考https://en.wikipedia.org/wiki/Function_point。——译者注

- **一味地追求速度**：“发布吧！不要花时间重构了！”你可以责怪工程经理不理解保持系统设计整洁的重要性吗？当然，可能看上去你的进度在一段时间内是快了些，但任由系统质量变差终会将你推入深渊。

  要勇敢说“不”，否则的话，就自己保持重构。将每几分钟整理一下代码作为TDD的一个环节。如果有人问为什么，你可以这样说：“这就是我作为负责任的专业人员的做事方法。”

  必须抓住每次测试提供的机会，代码中的问题会很快累积并开始拖慢开发速度。你甚至会觉得必须请求做一次*重构迭代*。不要这么做！非技术人员对重构是什么一无所知。他们会把你的请求理解为“程序员只是想玩玩罢了，我不会在此迭代中得到任何与业务有关的回报”。

问题是，无论怎样努力尝试，当新功能不能容易地加入到已有代码中时，你不可避免地要做些迭代，这会使功能的完成比预期要晚一些。如果发生这样的情况，就请求原谅并做细致的调查，看看怎样才能避免这些额外的开销。不要让这习惯性地发生。

6.5 结束语

在本章中，你学习了怎样确保软件维持高质量的设计，以便日后可以轻松地维护；学习了怎样在TDD的重构阶段应用简单设计理念；同时，还了解了持续、增量重构的重要性，以及会阻碍充分重构的一些因素。

不要就此停下，要确保对测试应用相似的设计理念，我们将在下一章中深入探讨这个主题。同时，你也要通过其他途径阅读关于面向对象设计的更多知识。你所了解的良好设计的所有知识将帮助你成功地开发系统。

第 7 章

高质量测试

7

7.1　开场白

你已经学习了怎样测试驱动开发，更重要的是，在上一章中学习了怎样利用TDD打磨系统设计。安全且持续的重构可以让产品代码保持活力。在这一章中，你将学习怎样设计测试，以便提升回报并避免其成为维护负担。

可通过下列核心理念学习怎样开发高质量的测试。

- FIRST助记符——审核测试的重要方法
- 一个测试一个断言——帮助限制测试大小的准则
- 测试抽象——保持测试可读性的核心原则

7.2　测试先行

想知道你编写的单元测试是一个好的测试吗？可以对照FIRST原则（由Brett Schuchert和Tim Ottinger提出）来审查。这个助记符可以提醒你TDD定义中的关键部分：测试先行。

FIRST可以分解为如下部分：

- F是快速（Fast）；
- I是独立（Isolated）；
- R是可重复（Repeatable）；
- S是自我验证（Self-verifying）；
- T是及时（Timely）。

7.2.1　快速

通过规定、构建、重构的核心周期，TDD支持增量和迭代的开发方式。每个周期应该多长呢？越短越好。当代码不工作或破坏了其他功能时，你需要马上知道情况。在引入一个缺陷到发现它

之前，生成的代码越多，找到并修复该缺陷所耗费的时间就越长。你需要超快速的反馈！

编写代码时，我们都会犯错。初写的代码很难达到很好的设计，就像作家写初稿一样，第一阶段编写的代码往往显得粗糙。但胡乱堆砌的代码会越来越难以修改。最可能让代码回到正轨的做法是什么呢？持续地检查、整理每一点代码。

必须确保所修改的或新加的测试可以通过，且所做的改动不会破坏系统中的其他代码。每做一次小的改动，就要运行一下所有的单元测试。

理想情况下，在获得反馈前，要编写一点点代码（如一两行）。但这样做会增加编译、链接和运行测试集的开销。

保持周期的低开销有多重要呢？如果平均需要三到四秒来编译、链接和运行测试的话，那么增量开发只需要很少的代码就能得到很多反馈。但试想一下，如果测试集需要花费两分钟来构建和运行，你会多久运行一次呢？或许每十分钟或十五分钟一次？如果运行一遍测试需要二十分钟，恐怕一天也就只能运行几遍。

如果没有快速反馈，你编写的测试会减少，代码重构会更少，从引入问题到发现问题的间隔也会变长。依赖以往的结果意味着，只能获得很少的TDD的潜在益处。这时，你也可能会选择放弃TDD。不要成为这样的人！

1. 构建的开销

C++系统的构建时间是一个很大的问题。在大型系统中编译、链接可能需要好几分钟，有时要更长时间。

大部分构建时间与代码的依赖结构直接相关。依赖于改动的代码必须重新编译。

要想很好地践行TDD，需要打造一个最小化大量重编译的设计。如果一个常用的类暴露了大量的接口，那么接口变化时必须重新编译使用此类的客户端，即便你的改动和它们对这个类的使用没有交集。按照*接口隔离原则*（Interface Segregation Principle，ISP，《敏捷软件开发：原则、模式与实践》），迫使客户端程序依赖一个它们不用的接口意味着存在设计缺陷。

同样，滥用其他原则会导致编译时间过长。*依赖倒置原则*（Dependency Inversion Principle，DIP）让你依赖抽象，而非实现细节（《敏捷软件开发：原则、模式与实践》）。如果改变了具体类的细节，那么必须重新编译所有客户端程序。

可以引入一个接口——一个纯虚空类，由一个具体类来实现。如果改动具体类的实现细节，客户端程序通过接口提供的抽象与其交互，且不会触发客户端程序的重编译。

如果在重构的过程中引入了新的私有方法，你就会发现，重新构建会耗时很久，这会让人失去耐心。或许可以考虑使用“指针实现”（pointer to implementation，PIMPL）惯用法。为了使用PIMPL，要将具体的实现细节提取成一个独立的实现类。将来自接口的调用委托给这个实现类。然后就可以自由地改变实现、随心所欲地创建新的函数了，而不用重新编译依赖公开接口的代码。

7

有了TDD后，你的设计抉择不再是模糊不清的思虑；这些抉择直接关系到你能不能成功。用TDD获得成功的诀窍是，保持整洁、高效。

2. 对协作者的依赖

依赖改动会增加编译时间。为了运行这些测试，对依赖的担忧开始发生转变：在测试其他代码时产生的依赖增加了测试的执行时间。

如果测试代码和另外一个类交互，且这个类必须调用外部API（如一个数据库调用），那么测试必须等待API调用完成。（此时，测试变成了集成测试，而非单元测试）花几毫秒建立数据库连接并执行一次查询，似乎不会太浪费时间。但是，如果拥有几千个测试的测试集中的大部分测试都有此开销，那么就要花费几分钟或更长时间才能运行完整个测试集。

3. 运行一个测试子集

大部分单元测试工具允许运行整个测试集的一个子集。例如，Google Test可以指定一个过滤器。举例来说，下面的命令行给测试传递了一个过滤器，使所有fixture名称以Holding开头的测试或名称中包含Avail的测试得以运行。运行更小的测试子集能节省一些执行时间。

```
./test --gtest_filter=Holding*.*Avail*
```

能这样做不代表应该这么做……至少不该习惯性地使用。习惯性地过滤掉一些测试，说明存在更大的问题——测试依赖太多的慢速因素。先处理真正的问题！

当不能容易地运行所有测试时，不要立刻一次只运行一个测试。要找到一次运行尽可能多的测试的方法。至少在一次运行一个测试前，尝试在特定fixture（如`Holding*.*`）下运行所有的测试。

在初期运行测试的子集或许能节省点时间，但请记住，运行的测试越少，今后发现的问题就会越多，进而也就需要更多时间来进行修复。

7.2.2　独立

使用TDD时，每个测试至少应该失败一次，那么在创建新测试时，你就会知道失败的原因。但若隔了三天或三个月呢？测试失败的原因还会清晰吗？对你或其他需要找出问题根源的人来说，创建一个可以由多个原因导致失败的测试是很浪费时间的。

测试应该是独立的，即只会因为一个原因失败。小而集中的测试可以驱动开发出一小段行为，从而增加了独立性。

同时，每个测试应该验证一小段独立于外部因素的逻辑。如果测试所验证的代码要与数据库、文件系统或其他的API交互，那么导致测试失败的因素可以是很多种。引入测试替身（参见第5章）可以获得独立性。

测试不仅要独立于产品系统的外部因素，还要彼此独立。使用静态数据的所有测试都可能因为旧的数据而失败。

如果测试需要大量的设置，或产品代码可能保持旧的数据，你会发现，深入调查才能找到导致测试失败的细微改动。可以引入前置条件断言来验证测试在Arrange阶段的所有假设。

c7/2/libraryTest/HoldingTest.cpp

```
TEST_F(ACheckedInHolding, UpdatesDateDueOnCheckout)
{
➤   ASSERT_TRUE(IsAvailableAt(holding, *arbitraryBranch));
    holding->CheckOut(ArbitraryDate);
    ASSERT_THAT(holding->DueDate(),
       Eq(ArbitraryDate + date_duration(Book::BOOK_CHECKOUT_PERIOD)));
}
```

图书馆应用程序

本章的代码示例来自于一个小型演示图书馆系统。一些关键术语的定义可以帮助你更好地理解示例。**顾客**可以从**分馆**中签出或租借**藏书**。**顾客**是一个人；分馆是图书系统中的一个物理位置；**藏书**是图书馆中的书的副本。

如果前置条件断言失败，你会少浪费些时间查找、修复问题。但是，如果发现要经常使用这一技巧，还是找一个简化设计的方法吧——前置条件断言意味着你对系统的理解还不够，也意味着你在设置阶段隐藏太多的信息。

7.2.3 可重复

高质量的单元测试是可重复的，即可以一遍遍地运行，并且总是获得相同的结果，无论其他测试（如果有的话）是否先运行。测试集的快速反馈能够提供很多东西，有时我会再运行一次，只是为了得到测试全部通过的满足感。有时，虽然之前的一次测试通过了，接下来的运行却是失败的。

间歇性的测试失败不是一件好事。这意味着一定程度上的不确定性或测试运行中存在异常行为。找到异常行为的原因可能需要大量的精力。

下列原因可能导致测试的间歇性失败。

- **静态数据**：好的单元测试不会依赖其他测试的影响，同样，也不会让残留的状态造成问题。如果静态数据可能使测试失败，那么你在加入新测试或移除一些测试时，才可能看到真正的失败。在一些单元测试框架中，测试被加入到基于哈希的集合中，这意味着测试的执行顺序会随着测试数量的变化而改变。

- **不稳定的外部服务**：避免编写依赖于你控制不了的外在因素的单元测试，如当前时间、文件系统、数据库以及其他一些API调用。可在必要时引入测试替身（参见第5章）来打破这种依赖关系。
- **程序并发**：多线程或其他多道执行技术会引入一些不确定的行为，这对单元测试而言可能是极大的挑战。参见第9章来获得有关测试驱动开发多线程代码的建议。

7.2.4 自我验证

自动化测试解放了你自己——淘汰了慢速且有风险的手动测试。单元测试必须执行代码并验证代码能否在你不参与的情况下自动工作。一个单元测试至少有一个断言。它必须在其存在的周期里至少失败一次，未来也必须有一些方法让它失败。

不要在这条准则上有任何退让。不要在测试中加入`cout`语句来替代断言。手动的控制台验证或日志文件输出既浪费时间也增加风险。

投机取巧的程序员会发现，编写没有断言的测试可以提升测试覆盖率（受误导的经理所追求的目标）。这些测试纯属浪费精力，但在不做任何断言的情况下执行大量的代码却改善了指标。

7.2.5 及时

什么时候编写测试？及时编写意味着你要先编写测试。为什么？因为你在做TDD，而且正是因为它是保持高质量代码库的最好方法，所以你才使用它。

同样，不要在编写代码前编写一堆测试。相反，每次只写一个测试，甚至在每个测试中一次只加入一个断言。要尽可能地增量化，将每个测试视作一小段规范，让你可以快速驱动一致的行为。

7.3 一个测试一个断言

在系统中测试驱动开发小而不相关联的行为。在TDD的每个周期中，你要规定行为以及验证此行为确实可行的方法——断言。

为了让以后的程序员理解系统涉及的行为，测试必须清楚地陈述意图。意图最重要的声明是测试名称，它应该澄清上下文和目标。

测试覆盖的行为越多，测试名就越不能准确地描述行为。

在图书馆系统中，藏本在顾客借出后就不可再借了，在顾客归还后恢复为可借状态。可设计一个专门用于可借性的测试。

c7/3/libraryTest/HoldingTest.cpp

```
TEST_F(HoldingTest, Availability)
{
   holding->Transfer(EAST_BRANCH);
   holding->CheckOut(ArbitraryDate);
   EXPECT_FALSE(holding->IsAvailable());

   date nextDay = ArbitraryDate + date_duration(1);
   holding->CheckIn(nextDay, EAST_BRANCH);
   EXPECT_TRUE(holding->IsAvailable());
}
```

将多个行为揉进一个测试，在一定程度上将它们绑在了一起。你可能发现许多方法会和其他方法一同使用，这意味着找出哪个方法是哪个测试所测是相当困难的。其缺点是，阅读测试的人需要花费更多的时间弄明白怎么回事，尤其是当你往一个测试中加入3个、4个或数不清的行为时。

分成多个测试使你能够用测试名清楚地表明在何种情况下发生了何种行为。同时，也能让你充分利用Arrange-Act-Assert/Given-When-Then来增加清晰度。

c7/3/libraryTest/HoldingTest.cpp

```
TEST_F(AHolding, IsNotAvailableAfterCheckout)
{
   holding->Transfer(EAST_BRANCH);

   holding->CheckOut(ArbitraryDate);

   EXPECT_THAT(holding->IsAvailable(), Eq(false));
}

TEST_F(AHolding, IsAvailableAfterCheckin)
{
   holding->Transfer(EAST_BRANCH);
   holding->CheckOut(ArbitraryDate);

   holding->CheckIn(ArbitraryDate + date_duration(1), EAST_BRANCH);

   EXPECT_THAT(holding->IsAvailable(), Eq(true));
}
```

单一目的测试的名称能自我澄清，即不需要阅读测试代码来了解它做了什么。这些测试名的完整集合正如系统能力的索引表。你开始将测试名称视作相关的行为组，而不仅仅是不相关的行为验证。

全面地查看测试名称能引发你思考一些缺失的测试。“我们有描述藏本借出和归还后的可用性的测试。在进货过程中向系统加入新书时，书籍可用性的哪些量要设置为真呢？最好编写一个测试！”

你曾经有多于一个断言的测试吗？要尽量保证只有一个。但多于一个有时也是合理的。

断言是后置条件。如果一个行为需要多个断言，可在测试中引入。想想`IsEmpty()`，通常是为了增加表达力而引入它，而这一表达力是`Size()`这类函数所不具备的。你可能会在判断一个新集合是否为空时，选择同时使用这两个函数。

同时，也可能在验证一大堆数据时引入多个断言。

c7/3/libraryTest/HoldingTest.cpp

```
TEST_F(AHolding, CanBeCreatedFromAnother)
{
   Holding holding(THE_TRIAL_CLASSIFICATION, 1);
   holding.Transfer(EAST_BRANCH);

   Holding copy(holding, 2);

   ASSERT_THAT(copy.Classification(), Eq(THE_TRIAL_CLASSIFICATION));
   ASSERT_THAT(copy.CopyNumber(), Eq(2));
   ASSERT_THAT(copy.CurrentBranch(), Eq(EAST_BRANCH));
   ASSERT_TRUE(copy.LastCheckedOutOn().is_not_a_date());
}
```

最后，在行为的描述不再变化的前提下，越来越多的数据加入后，实现变得越来越特定化时，你可以合并断言。例如，以下的实用类将罗马数字转换成了阿拉伯数字。

c7/3/libraryTest/RomanTest.cpp

```
TEST(ARomanConverter, AnswersArabicEquivalents)
{
   RomanConverter converter;
   ASSERT_EQ("I", converter.convert(1));
   ASSERT_EQ("II", converter.convert(2));
   ASSERT_EQ("III", converter.convert(3));
   ASSERT_EQ("IV", converter.convert(4));
   ASSERT_EQ("V", converter.convert(5));
   // ...
}
```

对于这些情况，可以将测试拆分开。但将测试拆分显得没有太大意义，可以用能想到的测试名来说明：ConvertsRomanIIToArabic、ConvertsRomanIIIToArabic等。又或者CopyPopulates-Classification、CopyPopulatesCopyNumber等。

要谨记一个关键的准则——一个测试只有一个行为。如果还不够清楚，那么可以这样认为，任何拥有条件逻辑（如`if`语句）的测试几乎都与此准则相悖。

一个测试一个断言不是硬性原则，但通常是更好的选择。要努力让一个测试有更少的断言，而不是更多。越是这样做，越会发现其价值所在。就目前来说，尽量让一个测试只有一个断言，然后思考结果。

7.4 测试抽象

Bob大叔将抽象定义为“去粗取精”[①]。抽象在测试和面向对象设计中同等重要。因为你要将测试当作文档来读，所以它们必须正中其意，以最清晰、最简洁的方式声明其意图。

简单地讲，可以通过下面几个途径增加测试的抽象度：使它们更加内聚（一个测试一个断言）；更好地命名（测试自身及其内部的代码）；抽象掉多余的东西（可利用fixture辅助函数或`SetUp()`）。

下面来看一下九种不同的测试坏味，并相应地整理测试代码。

7.4.1 臃肿的初始化

从LineReader类的一个测试开始，该测试名没有过多说明LineReader是怎样工作的。希望测试的整理工作能够帮到我们。

c7/3/linereader/LineReaderTest.cpp

```
TEST(LineReaderTest, OneLine) {
➤ const int fd = TemporaryFile();
➤ write(fd, "a", 1);
➤ lseek(fd, 0, SEEK_SET);
  LineReader reader(fd);

  const char *line;
  unsigned len;
  ASSERT_TRUE(reader.GetNextLine(&line, &len));
  ASSERT_EQ(len, (unsigned)1);
  ASSERT_EQ(line[0], 'a');
  ASSERT_EQ(line[1], 0);
  reader.PopLine(len);

  ASSERT_FALSE(reader.GetNextLine(&line, &len));

  close(fd);
}
```

测试的前三行创建了一个临时文件，并在文件里写入了一个字符（`"a"`），然后将文件指针指向文件头部。这个臃肿的构造需要阅读测试的人看一些无关紧要的测试设置细节。可以用一行抽象来替代它。

c7/4/linereader/LineReaderTest.cpp

```
TEST(LineReaderTest, OneLine) {
➤ const int fd = WriteTemporaryFile("a");
  LineReader reader(fd);

  const char *line;
```

① “粗”在这里表示不相关的部分，而“精”则表示关键的部分。——译者注

```
    unsigned len;
    ASSERT_TRUE(reader.GetNextLine(&line, &len));
    ASSERT_EQ(len, (unsigned)1);
    ASSERT_EQ(line[0], 'a');
    ASSERT_EQ(line[1], 0);
    reader.PopLine(len);

    ASSERT_FALSE(reader.GetNextLine(&line, &len));

    close(fd);
}
```

测试变短了几行，而且隐藏了创建一个有少量数据的文件的实现细节。一般不需要关心这个，极少数需要的情况下，可以浏览来了解`WriteTemporaryFile()`到底做了什么。

7.4.2　不相关的细节

这个测试首先创建了一个临时文件。如果最初写测试的程序员是一个优秀的编码人员，他应确保在测试末尾关闭文件。

c7/5/linereader/LineReaderTest.cpp

```
TEST(LineReaderTest, OneLine) {
➤   const int fd = WriteTemporaryFile("a");
    LineReader reader(fd);

    const char *line;
    unsigned len;
    ASSERT_TRUE(reader.GetNextLine(&line, &len));
    ASSERT_EQ(len, (unsigned)1);
    ASSERT_EQ(line[0], 'a');
    ASSERT_EQ(line[1], 0);
    reader.PopLine(len);

    ASSERT_FALSE(reader.GetNextLine(&line, &len));

➤   close(fd);
}
```

对`close()`的调用显得有点乱，这是在理解测试时让人分心的另一个实现细节。可以利用`TearDown()`确保文件关闭。同时，也可以将变量`fd`（文件描述符）声明的类型信息移除，将其移进fixture。

c7/6/linereader/LineReaderTest.cpp

```
class LineReaderTest: public testing::Test {
public:
➤    int fd;
     void TearDown() {
➤       close(fd);
     }
```

```
};

TEST_F(LineReaderTest, OneLine) {
➤  fd = WriteTemporaryFile("a");
   LineReader reader(fd);

   const char *line;
   unsigned len;
   ASSERT_TRUE(reader.GetNextLine(&line, &len));
   ASSERT_EQ(len, (unsigned)1);
   ASSERT_EQ(line[0], 'a');
   ASSERT_EQ(line[1], 0);
   reader.PopLine(len);

   ASSERT_FALSE(reader.GetNextLine(&line, &len));
}
```

现在似乎很少用到这个临时变量了。我们将LineReader实例的创建写成一行。

c7/7/linereader/LineReaderTest.cpp

```
TEST_F(LineReaderTest, OneLine) {
➤  LineReader reader(WriteTemporaryFile("a"));

   const char *line;
   unsigned len;
   ASSERT_TRUE(reader.GetNextLine(&line, &len));
   ASSERT_EQ(len, (unsigned)1);
   ASSERT_EQ(line[0], 'a');
   ASSERT_EQ(line[1], 0);
   reader.PopLine(len);

   ASSERT_FALSE(reader.GetNextLine(&line, &len));
}
```

嗯……还有点小问题。不再在`TearDown()`中关闭这个临时文件（相反，试图关闭一个未初始化的文件描述符`fd`），我们选择通过支持RAII来改善LineReader的设计，并在析构函数中自行关闭文件。（要想更深入地了解RAII惯用法，请参见http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization。）代码的实现细节留给读者！（或者你可以看一下提供的示例源代码。）

测试还包含了一些大多数时候不需要看到的细节——声明`line`和`len`变量的两行代码。同时，它们在几个LineReader的测试中重复出现。让我们去掉这些重复代码吧！

c7/8/linereader/LineReaderTest.cpp

```
class LineReaderTest: public testing::Test {
public:
    int fd;
➤   const char *line;
➤   unsigned len;
};
```

```
TEST_F(LineReaderTest, OneLine) {
  LineReader reader(WriteTemporaryFile("a"));

  ASSERT_TRUE(reader.GetNextLine(&line, &len));
  ASSERT_EQ(len, (unsigned)1);
  ASSERT_EQ(line[0], 'a');
  ASSERT_EQ(line[1], 0);
  reader.PopLine(len);

  ASSERT_FALSE(reader.GetNextLine(&line, &len));
}
```

7.4.3　缺失的抽象

许多开发者经常忽视创建简单抽象的契机。尽管为获得有待商榷的收益似乎付出了额外的精力，但将小段代码提取为辅助函数和类却是三赢的。第一，它增强了代码的表达力，潜在地消除了解释性注释的需求。第二，它提升了重用这些代码的机会，也有助于消除大量的重复代码。第三，它使接下来的测试更易于编写。

目前的测试需要三行代码来验证从reader中读取下一行的结果。

c7/9/linereader/LineReaderTest.cpp

```
TEST_F(LineReaderTest, OneLine) {
  LineReader reader(WriteTemporaryFile("a"));

  ASSERT_TRUE(reader.GetNextLine(&line, &len));
➤ ASSERT_EQ(len, (unsigned)1);
➤ ASSERT_EQ(line[0], 'a');
➤ ASSERT_EQ(line[1], 0);
  reader.PopLine(len);

  ASSERT_FALSE(reader.GetNextLine(&line, &len));
}
```

一个辅助函数可以将三个断言转为一个更加抽象的声明。（如果使用的单元测试工具支持，也可以引入一个基于匹配器的自定义断言。）

c7/10/linereader/LineReaderTest.cpp

```
void ASSERT_EQ_WITH_LENGTH(
   const char* expected, const char* actual, unsigned length) {
   ASSERT_EQ(length, strlen(actual));
   ASSERT_STREQ(expected, actual);
}

TEST_F(LineReaderTest, OneLine) {
  LineReader reader(WriteTemporaryFile("a"));

  ASSERT_TRUE(reader.GetNextLine(&line, &len));
```

```
➤  ASSERT_EQ_WITH_LENGTH("a", line, len);
   reader.PopLine(len);

   ASSERT_FALSE(reader.GetNextLine(&line, &len));
}
```

7.4.4 多重断言

我们已经将测试精简到只包含几行语句和三个断言。现在用一个测试一个断言原则来创建三个测试，每一个测试都要有清晰的名字来概括其意图。

c7/11/linereader/LineReaderTest.cpp

```
TEST_F(GetNextLinefromLineReader, UpdatesLineAndLenOnRead) {
  LineReader reader(WriteTemporaryFile("a"));
  reader.GetNextLine(&line, &len);
  ASSERT_EQ_WITH_LENGTH("a", line, len);
}

TEST_F(GetNextLinefromLineReader, AnswersTrueWhenLineAvailable) {
  LineReader reader(WriteTemporaryFile("a"));
  bool wasLineRead = reader.GetNextLine(&line, &len);
  ASSERT_TRUE(wasLineRead);
}

TEST_F(GetNextLinefromLineReader, AnswersFalseWhenAtEOF) {
  LineReader reader(WriteTemporaryFile("a"));
  reader.GetNextLine(&line, &len);
  reader.PopLine(len);
  bool wasLineRead = reader.GetNextLine(&line, &len);
  ASSERT_FALSE(wasLineRead);
}
```

审阅一下这些新测试及其名称，不难发现缺少了一些测试。如果不加分析和直觉判断，`PopLine()`的行为就不能得到充分的说明，我们也会思考如果连续调用两次`GetNextLine()`会发生什么。可以添加漏掉的测试：AdvancesToNextLineAfterPop和RepeatlyReturnCurrentRecord（这个就当作给读者的练习了）。

持续地审阅所有的测试名能够帮助你找到规范中的漏洞。

7.4.5 不相关的数据

测试中使用的数据有助于了解测试。内嵌在代码中的字面常量只会分散注意力，更糟糕的是，让人产生疑惑。如果一个函数调用需要参数，但这些参数与当前的测试无关，通常可以传入0或类似的数值来表示空（例如，字符串就用""表示）。对于阅读测试的人来说，这些数据应该提示“此数据无关”。（如果0是有意义的值，那就引入一个常量来帮助说明为什么。）

7

有时你别无选择，只能传递一个非0或非空的值。这时，一个简单的常量能迅速地告诉阅读人员他们需要知道的东西。在AnswersTrueWhenLineAvailable测试中，我们不关心文件内容，所以将传给`WriteTemporaryFile()`的字符`"a"`替换为意图明显的名称。

c7/12/linereader/LineReaderTest.cpp

```
TEST_F(GetNextLinefromLineReader, AnswersTrueWhenLineAvailable) {
➤ LineReader reader(WriteTemporaryFile(ArbitraryText));

  bool wasLineRead = reader.GetNextLine(&line, &len);

  ASSERT_TRUE(wasLineRead);
}
```

对臃肿的测试进行一些处理后，可以得到三个简洁的测试，每个就只有屈指可数的几行代码或者更少。每个测试都清晰易懂。现在，我们更加清楚LineReader的行为了。

为了嗅出更多代码坏味，我们来看看一些不够整洁的其他测试——LineReader的测试目前已经足够好了。

7.4.6 不必要的测试代码

有一些代码根本就不属于测试。本节讨论了一些可以从测试中完全移除的代码。

1. 断言不空

段错误可不是好玩的。解引用空指针是不会有好结果的，剩余的测试会因为崩溃而无法运行。防御性的编程自然是一个可以理解的应对方案。

c7/12/libraryTest/PersistenceTest.cpp

```
TEST_P(PersistenceTest, AddedItemCanBeRetrievedById)
{
    persister->Add(*objectWithId1);

    auto found = persister->Get("1");

➤   ASSERT_THAT(found, NotNull());
    ASSERT_THAT(*found, Eq(*objectWithId1));
}
```

但是请记住，之所以设计测试，是为了驱动开发常规行为或产生预想的失败。对于持久化[①]的代码来说，已经有一个测试演示了何时`Get()`会返回空（Null）。

c7/12/libraryTest/PersistenceTest.cpp

```
TEST_P(PersistenceTest, ReturnsNullPointerWhenItemNotFound)
```

① 这里的持久化就是将数据永久保存下来。——译者注

```
{
    ASSERT_THAT(persister->Get("no id there"), IsNull());
}
```

AddedItemCanBeRetrievedById是一个常规路径的测试。一旦测试可以工作，就应该一直工作……除非未来某人的代码引入了一个缺陷或内存分配失败。因此，对空的检查（ASSERT_THAT (found, NotNull())）在此常规路径下不大可能失败。

我们要移除这句代码。（应该杜绝使用裸指针以消除后患。）它对测试的文档化价值贡献并不大，仅仅是为了安全。移除这个保护语句的缺点是，一旦指针为空，就会产生段错误。我们也乐于对此作出权衡——虽然极少遇到，但最坏的情况是我们认可这个段错误，并加上一个空检查，然后重新运行测试来验证我们的猜想。如果测试运行得很快，那就没什么大问题。

如果你不想偶尔几次测试因段错误崩溃，一些单元测试框架提供了另一种方案，该方案不需要一个额外代码行来验证指针。例如，Google Test提供了匹配器Pointee()。

c7/13/libraryTest/PersistenceTest.cpp

```
TEST_P(PersistenceTest, AddedItemCanBeRetrievedById)
{
    persister->Add(*objectWithId1);

    auto found = persister->Get("1");

➤   ASSERT_THAT(found, Pointee(*objectWithId1));
}
```

在做TDD时，想象一下在增量步伐中为指针为空检查编写一个断言。这样做没什么问题。但是，一旦你认为测试完成了，要回头看一下代码，消除一些不具有文档化意义的代码。通常而言，这类不必要的代码不仅仅只有指针为空检查的断言。

7

2. 异常处理

如果开发的代码会产生异常，你需要编写一个测试来记录这个异常是怎么发生的。在图书馆系统中，增加分馆至少在一种情形下会抛出异常。

c7/13/libraryTest/BranchServiceTest.cpp

```
TEST_F(BranchServiceTest, AddThrowsWhenNameNotUnique)
{
   service.Add("samename", "");

   ASSERT_THROW(service.Add("samename", ""), DuplicateBranchNameException);
}
```

由于add()函数会抛出异常，一些程序员就想在调用add()的其他测试中做些保护措施。

c7/13/libraryTest/BranchServiceTest.cpp

```
TEST_F(BranchServiceTest, AddGeneratesUniqueId)
{
```

```
    // 不要这样做！
    // 从不会产生异常的测试里去掉try/catch
    // 不会产生异常
    try
    {
       string id1 = service.Add("name1", "");
       string id2 = service.Add("name2", "");
       ASSERT_THAT(id1, Ne(id2));
    }
    catch (...) {
       FAIL();
    }
}
```

你所设计的大部分测试是为了常规代码路径，它们是不会产生异常的。同样，如果向10个调用`add()`的其他测试中加入`try/catch`块，那么你就增加了60行异常处理代码。这些不必要的异常处理代码只会影响测试的可读性，并增加维护成本。

不被异常处理包裹后，相关的测试代码显得清洁多了。

c7/14/libraryTest/BranchServiceTest.cpp

```
TEST_F(BranchServiceTest, AddGeneratesUniqueId)
{
   string id1 = service.Add("name1", "");
   string id2 = service.Add("name2", "");

   ASSERT_THAT(id1, Ne(id2));
}
```

3. 断言失败的消息

并不是所有的单元测试框架都支持Hamcrest风格标记（`ASSERT_THAT`）。或者你的代码有点老旧，仍在使用经典的断言（例如，`ASSERT_TRUE`，参见第4章）。（或者你会发现Hamcrest没什么大的意义。）

Hamcrest风格的断言在改善错误信息方面有另外的好处。相比之下，如果简单的`ASSERT_TRUE`失败，其所产生的错误信息或许不能立马显示你想要知道的信息。像CppUnit这样的测试框架，允许你提供额外的参数来表达断言失败时该显示什么信息。

```
CPPUNIT_ASSERT_MESSAGE(service.Find(*eastBranch),
   "unable to find the east branch");
```

我的建议是去除这些断言失败的消息。为每个断言都加入这类信息会显得十分混乱，从而导致阅读测试变得困难，并增加需要维护的代码。阅读不带消息的断言反而很轻松。

```
CPPUNIT_ASSERT(service.Find(*eastBranch));
```

带有消息的断言不能带来任何有价值的东西。就像正常的代码注释一样，应该尽量避免此类需求。如果一个断言不带失败消息就没意义的话，那么还是先解决测试中的其他问题吧！

如果一个测试在测试集运行中意外失败了，也许不能从失败消息中立刻明确地看出原因。但通常要想找到导致失败的那行测试代码，需要得到所需信息。如果必要，可以加入临时的失败消息，然后再运行一遍。

4. 注释

如果必须加注释来说明测试是做什么的，那你已经忘了测试文档化的要领。重写测试吧，专注于更好的命名和内聚。

或许有时会看到类似如下的测试代码：

c7/15/libraryTest/BranchServiceTest.cpp

```
// 添加分支的测试，增加了计数
TEST_F(BranchServiceTest, AddBranchIncrementsCount)
{
   // 第一个分支
   service.Add(*eastBranch); // 东
   ASSERT_THAT(service.BranchCount(), Eq(1));

   // 第二个分支
   service.Add(*westBranch); // 西
   ASSERT_THAT(service.BranchCount(), Eq(2)); // count now 2
}
```

有些人觉得注释很有帮助，但注释不该将测试所做的（如果组织好的话就能清楚表明的）再说一遍。在保留测试表达力的前提下，努力找到一个去除注释的方法。对前一个测试而言，去除所有注释并不影响理解。

c7/16/libraryTest/BranchServiceTest.cpp

```
TEST_F(BranchServiceTest, AddBranchIncrementsCount)
{
   service.Add(*eastBranch);
   ASSERT_THAT(service.BranchCount(), Eq(1));
   service.Add(*westBranch);
   ASSERT_THAT(service.BranchCount(), Eq(2));
}
```

5. 隐含之意

“为什么测试这样断言其行为？”你可能希望，在不需要浪费时间分析测试或产品代码的前提下，测试阅读者就能够回答这个问题。

通常来说，你会将实现细节从测试移至`SetUp()`或另一个辅助函数。但注意，不要隐藏太多！否则测试阅读者需要看得更深入才能知道答案。合适的函数和变量名能使测试意图达到顾名思义的效果。

下面的测试需要在字里行间阅读一番：

c7/16/libraryTest/BranchServiceTest.cpp

```
TEST_F(ABranchService, ThrowsWhenDuplicateBranchAdded)
{
    ASSERT_THROW(service.Add("east", ""), DuplicateBranchNameException);
}
```

据我们推测，`SetUp()`中的代码在系统中加入了east这个分馆信息。或许此fixture中的所有测试都需要系统中已有一个分馆信息，因此，在`SetUp()`中添加east来消除重复代码是个好主意。但是，为什么要让测试者做额外的努力来理解测试呢？

可以通过改变其名称，引入一个有意义的fixture名称来阐明测试意图。

c7/17/libraryTest/BranchServiceTest.cpp

```
TEST_F(ABranchServiceWithOneBranchAdded, ThrowsWhenDuplicateBranchAdded)
{
    ASSERT_THROW(service.Add(alreadyAddedBranch->Name(), ""),
          DuplicateBranchNameException);
}
```

还有一个简单的测试，它需要测试阅读者跟进细节，并计算出两个日期间的天数。

c7/17/libraryTest/HoldingTest.cpp

```
TEST_F(AMovieHolding, AnswersDateDueWhenCheckedOut)
{
   movie->CheckOut(date(2013, Mar, 1));

   date due = movie->DueDate();

   ASSERT_THAT(due, Eq(date(2013, Mar, 8)));
}
```

在简单的表达式中加个断言可以极大地增加可读性。

c7/18/libraryTest/HoldingTest.cpp

```
TEST_F(AMovieHolding, AnswersDateDueWhenCheckedOut)
{
   date checkoutDate(2013, Mar, 1);
   movie->CheckOut(checkoutDate);
   date due = movie->DueDate();
   ASSERT_THAT(due, Eq(checkoutDate + date_duration(Book::MOVIE_CHECKOUT_PERIOD)));
}
```

将期待的测试输出和测试上下文建立联系是一门艺术。你要时刻具有创造力。同时要提醒自己了解测试的内部细节，但其他人则不需要。

6. 误导性的组织

一旦习惯了用Arrange-Act-Assert/Given-When-Then组织测试，你也会期待以此方式组织其他测试。而当看到组织方式与此不太一样的测试时，你的速度就会慢下来。因为你必须要努力弄清

楚哪些是测试的设置、哪些是真正的功能，所以就不能立即理解测试。找出下面这个测试（故意使用了无用的名称）的目的需要多长时间呢？

c7/18/libraryTest/HoldingServiceTest.cpp

```
TEST_F(HoldingServiceTest, X)
{
    HoldingBarcode barcode(THE_TRIAL_CLASSIFICATION, 1);
    string patronCardNumber("p5");
    CheckOut(barcode, branch1, patronCardNumber);
    date_duration oneDayLate(Book::BOOK_CHECKOUT_PERIOD + 1);
    holdingService.CheckIn(barcode.AsString(),
       *arbitraryDate + oneDayLate, branch2->Id());
    ASSERT_THAT(FindPatronWithId(patronCardNumber).FineBalance(),
       Eq(Book::BOOK_DAILY_FINE));
}
```

用AAA模式重写了下面的代码，以突出相关的部分：

c7/19/libraryTest/HoldingServiceTest.cpp

```
TEST_F(HoldingServiceTest, X)
{
    HoldingBarcode barcode(THE_TRIAL_CLASSIFICATION, 1);
    string patronCardNumber("p5");
    CheckOut(barcode, branch1, patronCardNumber);
    date_duration oneDayLate(Book::BOOK_CHECKOUT_PERIOD + 1);
    holdingService.CheckIn(barcode.AsString(),
          *arbitraryDate + oneDayLate, branch2->Id());
    ASSERT_THAT(FindPatronWithId(patronCardNumber).FineBalance(),
          Eq(Book::BOOK_DAILY_FINE));
}
```


将执行语句分开，可以很清楚地看出测试的重点在check-ins。进一步看这行代码，可以看出它涉及后来的check-ins。了解了系统中哪些部分会参与其中，可以快速地转到测试的断言部分，并判断出所验证的行为是一个客户的罚金，而且已经相应地更新了。

在软件开发中，理解现有代码所花费的时间是众多大开销之一。你所做的每件小事都有助于降低此开销，AAA的优点就是几乎不会带来任何开销。

7. 晦涩的命名

在软件设计中，良好的命名是你可以做的最重要的事情之一。通常而言，好的名称是测试关联问题的解决方案（参见7.4.6节）。同时你也会发现，如果不能赋予设计一个简洁的名称，很可能意味着该设计存在问题。

测试本身没什么不同。但如果一个测试的名称含糊不清或需要解释，那它就不能充当较好的文档。

c7/19/libraryTest/PersistenceTest.cpp

```
TEST_P(PersistenceTest, RetrievedItemIsNewInstance)
{
    persister->Add(*obj);

    ASSERT_FALSE(obj == persister->Get("1").get());
}
```

下面的简单改动会让阅读测试的人有不太一样的感觉。

c7/20/libraryTest/PersistenceTest.cpp

```
TEST_P(PersistenceTest, RetrievedItemIsNewInstance)
{
    persister->Add(*objectWithId1);

    ASSERT_FALSE(objectWithId1 == persister->Get("1").get());
}
```

不用对所有相关的数据命名，只要它能和一些事情很容易地联系起来即可。下面是一个示例：

c7/19/libraryTest/PatronTest.cpp

```
TEST_F(PatronTest, AddFineUpdatesFineBalance)
{
   jane->AddFine(10);

   ASSERT_THAT(jane->FineBalance(), Eq(10));
}
```

很明显，传递给AddFine()的参数10对应期望的罚金10。

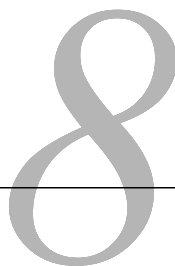
7.5　结束语

保持测试整洁、直接明了有助于你将它们作为文档以便日常使用。越是积极地阅读测试、总是先想到测试，就越会投入所需的精力来长期地维护它们。可以用本章提供的指导方针来帮助识别、纠正测试设计中的问题。

到目前为止，你已经学习了大量关于TDD的知识。你的产品代码是整洁的，并经过了充分的测试，测试代码也一样整洁并且很有裨益。但是对于那些不是你或你的队友开发的代码，该怎么办呢？“我好像曾经遇到过！”接下来，我们将探讨来自于遗留代码的挑战。

第 8 章

遗留代码的挑战



8.1 开场白

现在，你知道怎样用TDD方法编写设计良好的代码了。但对于大部分程序员而言，大部分时间并不是处理新的代码，而是要穿梭于大量的已有代码间，且这些代码是没有使用TDD的*遗留代码*。其中大部分是晦涩难懂、设计糟糕、构建仓促的代码。

该怎样应对这浩瀚如海的遗留代码呢？在这样一个代码库中仍然可以使用TDD吗？或者TDD方法只是适合原来的代码库吗？本章中介绍的一些技巧可以帮助你处理这些棘手且时刻存在的挑战。

你将在本章中学习一些在没有测试的情况下安全地重构代码的技巧和思路。你要向已有代码中加入一些测试来描述它们的行为特征，以便测试开发任何所需的改动。你也将学习怎样使用链接器存根，以快速摆脱第三方库经常对测试造成的令人头痛的问题。最后，你会学习到一种管理大规模代码重构的技巧——Mikado方法。

在本章的测试中，我们将CppUTest作为单元测试工具。CppUTest内建的模拟框架可以使处理遗留代码的测试更容易些。可根据自己的喜好继续使用Google Test/Google Mock，只需要针对单元测试代码做些相对简单的改动即可。

8.2 遗留代码

最资深的开发者在面对遗留代码时也会心生畏惧。试想要特化一个冗长且未经测试的函数中的一小部分。再想象一下，你要实现的功能是在30多行的代码间加入3行代码，且这3行代码要支持多态行为。作为经验老到的程序员，你知道合理的设计是将相同的行为重构至一个地方。因此，模板方法是一个可接受的方案。

（另外一种方案是引入条件分支。但当函数被各种各样的标志和内嵌的代码块困扰时，通常来说，模板方法才是应对代码逐渐退化的良方。）

但从经验来看，许多程序员对做正确的事情有点抵触，因为这会改变已有的、可以工作的代码。或许他们以前经历过此类改动带来的失败，可能还为此受到过责备。“如果没有坏，就不用修！”他们更倾向于复制、粘贴，然后修改。代码最终变成了60行而非少于35行。因此，代码库中的重复代码激增。

避免上述现象的最好方法是，使测试能够提供快速的反馈。这在大多数系统中似乎是奢侈品，遥不可及。Michael Feathers在《修改代码的艺术》中将遗留系统定义为缺少充足测试的系统。

与遗留代码库打交道会面临选择。你选择让维护成本不断增加，还是着手处理问题？若采取增量的方法并使用本章中列出的一些技巧，你会发现问题是有可能解决的。或许大多数时候还是值得一拼的，不允许任何人让系统变得更糟。不值得这么做的唯一情况是，你面对的是一个封闭的系统或即将被淘汰的系统。

8.3　法则

我们将看到一个增量地改善遗留代码库的示例。你会学到许多特别的技巧，每一个技巧都是为手头上特定的问题量身定制的。这些技巧是在《修改代码的艺术》及其他地方提到的几十个技巧的一个子集。你会发现这些技巧简单易学，并且一旦学会少量的技巧，就可以推导出剩下的许多技巧。

处理遗留代码问题的核心法则如下。

- 任何时候，只要可以就进行测试驱动。使用测试驱动方法时，有可能的话就将需要改动的代码作为新的成员或新的类。
- 不要让测试覆盖率缩水。非常容易出现的情况是，改动一点代码，然后认为这只是一些简单的代码行，并对之不予过多考虑。如果没有测试，每一行新加的代码都会导致测试覆盖率降低。
- 为了编写测试，必须改动现有代码！由于对协作对象的依赖，大多数情况下不能方便地为遗留代码编写测试。在编写测试前，需要一个打破依赖的方法。
- 可以在限制范围内实施微小的代码改动，这样会降低风险。用一些技巧就可以手动地做出一些小而安全的代码变换[①]。
- 你编写的每行代码都有风险，甚至一个敲错的字符都会引入潜在的缺陷，而这可能会浪费好几个小时。尽可能少写代码，每敲击一次键盘时都要想清楚。
- 坚持小的、增量的改动。这在TDD中是奏效的。同样，对遗留代码也是如此。步子迈得太大会使你陷入困境。
- 一次只做一件事。在处理遗留代码时，不要合并步骤或目标。例如，不要在重构的同时

① 代码变换不同于代码重构。——译者注

写测试。

- 有些增量的改动可能会使代码变得丑陋，要接受这一点。记住，一次只做一件事。可能需要几个小时做“正确”的事。不要等，现在就提交工作方案，因为这样或许就不用花费过多时间。

  同时，不要过于担心会违反一些设计原则。最终，你可能要抽出时间回过头来整理代码，或许你不会，但依然是进步了。你已经向正确的方向迈进了，而且也证明了所有代码依然可以工作。
- 增量地修改代码。面对庞大的代码库时，仅仅为遇到的相关代码编写测试或许不能带来巨大改观。更重要的是你内心深知任何新加入的东西都需要经过测试。

  秉承测试第一的心态，你会开始从加入的测试中获益。每通过一个测试，可以回顾一下测试覆盖的代码区域。几乎总能发现做一些小的、安全的重构工作的机会。同时，你也会发现在写完一个测试后，再写一个是如此容易。你可以在各个地方应用这种小的改善步伐而丝毫不影响产出，也将在困难重重的代码库中如履平地。

8.4 遗留应用程序

场景：WAV Snippet Publisher

作为一个内容转销商，我想要WAV Snippet Publisher从一个源文件目录下的每个WAV文件中抽出一段，然后生成一个新的WAV文件，并保存在目标目录下。

音频（Waveform Audio，WAV）文件标准出自于Microsoft和IBM，这基于他们的Resource Interchange File Format（RIFF），参见http://en.wikipedia.org/wiki/WAV。WAV文件包含音频数据，这些数据被编码成一个样本集合，通常用被广泛接受的脉冲码调制（Pluse-Code Modulation，PCM）标准，参见http://en.wikipedia.org/wiki/Pulse-code_modulation。可以在https://ccrma.stanford.edu/courses/422/projects/WaveFormat/找到WAV格式的一个简化版。

虽然实现上有诸多限制，但Snippet Publisher还是能够满足现有需求，但客户的需求是不断提高的。例如，它不能处理音频样本为奇数的情形，也不能处理多通道WAV文件。由于小端/大端的差异，它也不能支持所有的平台。客户要求我们除掉这些限制（这些留给读者用作以后的练习）。

场景：添加多通道的支持

目前，Snippet publisher仅能处理单通道WAV文件。作为内容转销商，要确保立体声效的WAV片段不会被中途截断。

啊！已经有变化了！不幸的是，Snippet Publisher几乎没有单元测试，但这也在预料之中。（在为本章准备时，我没有采用TDD来构建代码库。起初速度似乎比较快，但由于缺乏测试，花费的时间很快就比之前节省的时间还要多。）

open()函数包含了一大堆WAV Snippet Publisher处理逻辑。接下来的几页都是。

wav/1/WavReader.cpp

```
void WavReader::open(const std::string& name, bool trace) {
   rLog(channel, "opening %s", name.c_str());

   ifstream file{name, ios::in | ios::binary};
   if (!file.is_open()) {
      rLog(channel, "unable to read %s", name.c_str());
      return;
   }

   ofstream out{dest_ + "/" + name, ios::out | ios::binary};

   RiffHeader header;
   file.read(reinterpret_cast<char*>(&header), sizeof(RiffHeader));

   if (toString(header.id, 4) != "RIFF") {
      rLog(channel, "ERROR: %s is not a RIFF file",
         name.c_str());
      return;
   }
   if (toString(header.format, 4) != "WAVE") {
      rLog(channel, "ERROR: %s is not a wav file: %s",
         name.c_str(),
         toString(header.format, 4).c_str());
      return;
   }
   out.write(reinterpret_cast<char*>(&header), sizeof(RiffHeader));

   FormatSubchunkHeader formatSubchunkHeader;
   file.read(reinterpret_cast<char*>(&formatSubchunkHeader),
         sizeof(FormatSubchunkHeader));

   if (toString(formatSubchunkHeader.id, 4) != "fmt ") {
      rLog(channel, "ERROR: %s expecting 'fmt' for subchunk header; got '%s'",
         name.c_str(),
         toString(formatSubchunkHeader.id, 4).c_str());
      return;
   }

   out.write(reinterpret_cast<char*>(&formatSubchunkHeader),
         sizeof(FormatSubchunkHeader));

   FormatSubchunk formatSubchunk;
   file.read(reinterpret_cast<char*>(&formatSubchunk), sizeof(FormatSubchunk));

   out.write(reinterpret_cast<char*>(&formatSubchunk), sizeof(FormatSubchunk));

   rLog(channel, "format tag: %u", formatSubchunk.formatTag); // 显示十六进制文件?
   rLog(channel, "samples per second: %u", formatSubchunk.samplesPerSecond);
   rLog(channel, "channels: %u", formatSubchunk.channels);
   rLog(channel, "bits per sample: %u", formatSubchunk.bitsPerSample);
```

```
auto bytes = formatSubchunkHeader.subchunkSize - sizeof(FormatSubchunk);

auto additionalBytes = new char[bytes];
file.read(additionalBytes, bytes);
out.write(additionalBytes, bytes);

FactOrData factOrData;
file.read(reinterpret_cast<char*>(&factOrData), sizeof(FactOrData));
out.write(reinterpret_cast<char*>(&factOrData), sizeof(FactOrData));

if (toString(factOrData.tag, 4) == "fact") {
   FactChunk factChunk;
   file.read(reinterpret_cast<char*>(&factChunk), sizeof(FactChunk));
   out.write(reinterpret_cast<char*>(&factChunk), sizeof(FactChunk));

   file.read(reinterpret_cast<char*>(&factOrData), sizeof(FactOrData));
   out.write(reinterpret_cast<char*>(&factOrData), sizeof(FactOrData));

   rLog(channel, "samples per channel: %u", factChunk.samplesPerChannel);
}

if (toString(factOrData.tag, 4) != "data") {
   string tag{toString(factOrData.tag, 4)};
   rLog(channel, "%s ERROR: unknown tag>%s<", name.c_str(), tag.c_str());
   return;
}

DataChunk dataChunk;
file.read(reinterpret_cast<char*>(&dataChunk), sizeof(DataChunk));

rLog(channel, "riff header size = %u" , sizeof(RiffHeader));
rLog(channel, "subchunk header size = %u", sizeof(FormatSubchunkHeader));
rLog(channel, "subchunk size = %u", formatSubchunkHeader.subchunkSize);
rLog(channel, "data length = %u", dataChunk.length);

// TODO: 如果对这里有一个填充字节感到奇怪!
auto data = new char[dataChunk.length];
file.read(data, dataChunk.length);
file.close();

// 所有这些
//   out.write(data, dataChunk.length);
// TODO: 多个通道
➤ uint32_t secondsDesired{10};
➤ if (formatSubchunk.bitsPerSample == 0) formatSubchunk.bitsPerSample = 8;
➤ uint32_t bytesPerSample{formatSubchunk.bitsPerSample / uint32_t{8}};
➤ uint32_t samplesToWrite{secondsDesired * formatSubchunk.samplesPerSecond};
➤ uint32_t totalSamples{dataChunk.length / bytesPerSample};
➤
➤ samplesToWrite = min(samplesToWrite, totalSamples);
➤
➤ uint32_t totalSeconds{totalSamples / formatSubchunk.samplesPerSecond};
➤ rLog(channel, "total seconds %u ", totalSeconds);
```

```
➤
➤    dataChunk.length = samplesToWrite * bytesPerSample;
➤    out.write(reinterpret_cast<char*>(&dataChunk), sizeof(DataChunk));

➤    uint32_t startingSample{
➤       totalSeconds >= 10 ? 10 * formatSubchunk.samplesPerSecond : 0};
➤    rLog(channel, "writing %u samples", samplesToWrite);
➤    for (auto sample = startingSample;
➤         sample < startingSample + samplesToWrite;
➤         sample++) {
➤       auto byteOffsetForSample = sample * bytesPerSample;
➤       for (uint32_t byte{0}; byte < bytesPerSample; byte++)
➤          out.put(data[byteOffsetForSample + byte]);
➤    }
     rLog(channel, "completed writing %s", name.c_str());
     descriptor_->add(dest_, name,
           totalSeconds, formatSubchunk.samplesPerSecond, formatSubchunk.channels);
     out.close();
  }
```

open()函数里有注释、to-do、注释掉的处理逻辑、有问题的命名、魔数、重复代码，以及能解决所有问题的一站式代码。还有什么不喜欢的？

好吧，我们不是很喜欢这段代码。在尝试加入多通道的支持时，若基于这样错综复杂的代码，把事情搞砸的概率是很高的。为所要改动的代码加上测试有助于添加多通道的支持。

8.5　保持测试驱动开发的心态

在应对遗留代码时依然是测试先行。即使已经编写完测试所涵盖的代码，仍需要写测试来描述已有行为的特征。同时，也要以TDD的方式添加新代码。

你可能会意识到在事实发生后编写测试（有时候我称之为开发后测试，即Test-After-Development，TAD），会比用TDD编写代码更花费精力。究其主要原因是，如果程序员不考虑测试，那他们就不会用易于测试的方式组织代码；在TDD过程中，应该持续地重构，使代码元素[①]趋于更小化、更具可重用性，这有助于更容易地编写新的测试和代码。

测试open()函数看上去可能非常耗时。我们需要创建或找到一个甚至多个WAV文件以便测试之用，也要读取结果文件并检查其内容。目前我们没有时间处理这些细节。

生活可以更简单。只需要测试即将被改动的代码，参考底线会告诉你什么时候破坏了已有的行为。虽然你要判定出哪些依赖代码可能会因此被破坏，但不需要测试超出改动范围的代码。

同时，要避免那些必须与文件系统直接打交道的测试，以便保持测试集快速运行并减少管理文件的烦恼。尽量使用存储于内存的流而非文件流，这样可以部分达成效果。

---

① 这里的代码元素主要指一段逻辑内聚的代码，从前文可以看出，将这些代码提炼为小方法是一个很好的实践。
——译者注

8.6 支持测试的安全重构

为了开始增加对多通道的支持，先找出open()函数中有用的功能验证点吧！

函数的末尾似乎有一系列计算——总秒数、要写出的采样数目、开始的地方，等等。紧接着计算的是一个for循环，看上去要将采样写入输出文件中（这段代码在前面的open()函数代码列表中被标记出来了）。需要改变一两个计算部分并修改循环来支持多通道。

最有趣的代码是循环。我们来给它编写一些测试吧……但是怎样编写呢？为了达到open()函数的这个点，我们需要配置大量的信息。

与之相反，将循环代码隔离出来，放进一个成员函数中，然后直接测试即可。

提问：要改动代码了吗？是不是有破坏代码的风险？

回答：是的。为了覆盖必须修改的代码，我们需要添加测试。将一段代码提取成一个方法是我们可以采取的比较安全的代码变化方法。

提问：提取出一个函数，并将函数原型放进头文件会增加工作量。有没有更容易的方法呢？

回答：目前最简单的方法是方法提取。用测试覆盖open()函数的前半部分需要数小时。

提问：我现在理解并接受了为什么可测试的代码更好。但是你将方法声明为公有的，许多资深程序员都会鄙视这样的做法。

回答：如果真的需要，可以采用其他稍微保守点的技巧。例如，可以将方法定义为受保护的，创建一个派生的测试，将其定义为公有的。这需要许多额外的工作量和维护成本，而收益甚微。我倾向于更简单的方法。

提醒你的同事，知道代码正常工作更重要，不要为了不大可能出现的、随意暴露代码的行为杞人忧天。

同时，可以写一些审慎的注释，解释暴露函数的原因来缓解他们的不安。将成员暴露出来突显出了设计上的缺陷——依恋情结（feature envy[1]）。成员暴露后会迫使一些人修复这个设计问题。

这种函数提取方法会以小而机械的步骤完成。

(1) 先在待提取函数的地方录入函数调用。

wav/2/WavReader.cpp

```
uint32_t startingSample{
   totalSeconds >= 10 ? 10 * formatSubchunk.samplesPerSecond : 0};
```

① feature envy是众多代码坏味中的一个，它违背了将数据和其操作封装在一起的原则。参见《重构》一书。
——译者注

```
➤ writeSamples(out, data, startingSample, samplesToWrite, bytesPerSample);

  rLog(channel, "writing %u samples", samplesToWrite);
  for (auto sample = startingSample;
      sample < startingSample + samplesToWrite;
      sample++) {
    auto byteOffsetForSample = sample * bytesPerSample;
    for (uint32_t byte{0}; byte < bytesPerSample; byte++)
       out.put(data[byteOffsetForSample + byte]);
  }
  rLog(channel, "completed writing %s", name.c_str());
  descriptor_->add(dest_, name,
        totalSeconds, formatSubchunk.samplesPerSecond, formatSubchunk.channels);
```

(2) 在`open()`函数前加入相应的函数声明，简单地从代码中的函数调用复制/粘贴即可。（一旦通过编译，就将其从文件头部移除）补全函数的返回值类型（`void`）和函数体大括号。目前先使函数保持自由，这样在得到正确参数前，就不用在代码原型中重复此动作。

wav/2/WavReader.cpp

```
void writeSamples(out, data, startingSample, samplesToWrite, bytesPerSample) {
}
```

(3) 为`writeSamples()`的参数加上类型信息。利用编译器就能得知，调用`writeSamples()`时参数是否能与函数声明匹配，然后修复编译器指出的地方。一切完成后，将函数原型写进`WavReader.h`中，并将`writeSamples()`的实现限定在类内（WavReader::）。

wav/3/WaveReader.cpp

```
void WavReader::writeSamples(ofstream& out, char* data,
      uint32_t startingSample,
      uint32_t samplesToWrite,
      uint32_t bytesPerSample) {
}
```

wav/3/WavReader.h

```
public:
   // ...
   void writeSamples(std::ofstream& out, char* data,
         uint32_t startingSample,
         uint32_t samplesToWrite,
         uint32_t bytesPerSample);
```

(4) 将`for`循环移至`writeSamples()`内，并确保编译通过。有可能会编译失败，例如，忘了传入必要的参数。

wav/4/WavReader.cpp

```
void WavReader::writeSamples(ofstream& out, char* data,
      uint32_t startingSample,
      uint32_t samplesToWrite,
      uint32_t bytesPerSample) {
```

```
   rLog(channel, "writing %i samples", samplesToWrite);

   for (auto sample = startingSample;
       sample < startingSample + samplesToWrite;
       sample++) {
     auto byteOffsetForSample = sample * bytesPerSample;
     for (uint32_t byte{0}; byte < bytesPerSample; byte++)
        out.put(data[byteOffsetForSample + byte]);
   }
}
```

移除open()成员函数中的for循环。

wav/4/WavReader.cpp

```
uint32_t startingSample{
   totalSeconds >= 10 ? 10 * formatSubchunk.samplesPerSecond : 0};

➤ writeSamples(out, data, startingSample, samplesToWrite, bytesPerSample);

rLog(channel, "completed writing %s", name.c_str());

descriptor_->add(dest_, name,
      totalSeconds, formatSubchunk.samplesPerSecond, formatSubchunk.channels);
```

接下来编译并运行已有的测试。我们已经成功并安全地将一段代码提取为函数了。提取writeSamples()函数也会增强open()函数的抽象度。

有时，在尝试提取函数时会遇到棘手的编译错误。如果你觉得必须要改变代码才能使编译通过，多半是因为提取了错误的代码。停下来重新思考一下使用的方法。可以改变提取代码的范围，或先找到写更多测试的方法。

每从已有代码中复制一次就会引入重复代码。因此，许多测试驱动开发者认为复制/粘贴是“邪恶的”方法。在编写新代码时，应当维护记录每次粘贴的栈，并要记得通过重构掉所有的重复代码来弹清此栈。

就遗留代码来说，复制/粘贴通常是很好的方法。在操作已有代码时，最小化实际的编码量，从已有的结构中复制可以降低犯低级错误的风险。这里所采取的提取writeSamples()的步骤（只展现了一种可行的方法）最小化了代码输入量、最大化了编译器帮助抓住错误的能力。

我们没有改变writeSamples()中的任何代码。这就意味着唯一的风险在于是否正确地调用了函数。（如果函数不返回空，还要确保正确地处理了返回值。）如果不特意修改writeSamples()，也就不用为之编写测试。相反，我们将围绕传给writeSamples()的参数编写测试。我们会在后文中做这件事。

8.7 添加测试刻画已有行为

我们需要改动writeSamples()中的代码。从一些测试开始吧！

wav/6/WavReaderTest.cpp

```
#include "CppUTest/TestHarness.h"
#include "WavReader.h"
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

TEST_GROUP(WavReader_WriteSamples)
{
   WavReader reader{"",""};
   ostringstream out;
};

TEST(WavReader_WriteSamples, WritesSingleSample) {
   char data[] { "abcd" };
   uint32_t bytesPerSample { 1 };
   uint32_t startingSample { 0 };
   uint32_t samplesToWrite { 1 };
   reader.writeSamples(&out, data, startingSample, samplesToWrite, bytesPerSample);
   CHECK_EQUAL("a", out.str());
}

TEST(WavReader_WriteSamples, WritesMultibyteSampleFromMiddle) {
   char data[] { "0123456789ABCDEFG" };
   uint32_t bytesPerSample { 2 };
   uint32_t startingSample { 4 };
   uint32_t samplesToWrite { 3 };

   reader.writeSamples(&out, data, startingSample, samplesToWrite, bytesPerSample);
   CHECK_EQUAL("89ABCD", out.str());
}
```

针对writeSamples()的测试有助于我们单独理解循环的行为。而且，测试也不是很大。需要仔细阅读WriteMultibyteSampleFromMiddle测试，并要求掌握一点点数学知识。怎样能使之变得更直观呢？

为了避免使用实际文件而降低测试速度，我们使用快速的、存储于内存的字符串对象，其类型为std::ostringstream。这就要修改writeSamples()的接口，使其接受std::ostream对象，std::ofstream和std::ostringstream都继承自这个类型。更确切地说，我们修改接口以便其接受一个指向std::ostream的指针。产品代码可以传递文件流的地址；测试则可以传递字符串流的地址。改变writeSamples()中使用局部变量out的解引用方式，使其使用指针语义。

wav/6/WavReader.cpp

```
writeSamples(&out, data, startingSample, samplesToWrite, bytesPerSample);
```

wav/6/WavReader.cpp

```
➤ void WavReader::writeSamples(ostream* out, char* data,
        uint32_t startingSample,
        uint32_t samplesToWrite,
        uint32_t bytesPerSample) {
     rLog(channel, "writing %i samples", samplesToWrite);

     for (auto sample = startingSample;
          sample < startingSample + samplesToWrite;
          sample++) {
        auto byteOffsetForSample = sample * bytesPerSample;
        for (uint32_t byte{0}; byte < bytesPerSample; byte++)
➤          out->put(data[byteOffsetForSample + byte]);
     }
  }
```

8.8　被遗留代码转移注意力

测试集失败了！对于每个测试，CppUTest将执行开始和执行结束后的内存对比，如果发现不匹配就报失败。稍微考察一下代码，我们发现，内存泄漏问题的根源要么出在第三方日志库rlog，要么出在WavReader对rlog的使用上。处理内存泄漏不是主要目的。我们需要找出一条路继续前行。

CppUTest允许我们关闭内存泄漏检测，但这是一个非常有用的功能，值得保留。此外，就算关闭内存泄漏检测，每次运行测试时，贯穿WavReader代码之间的日志代码也会向控制台发送消息，这是十分令人讨厌的。除了默默忍受还有其他办法吗？在不修改代码的前提下关掉日志也是可行的，但已经没有时间了，我们需要继续前行。

第三方库会为测试制造无尽的烦恼。它们要么速度慢，要么需要大量的配置工作，还可能有副作用。如果你是第一次面对遗留代码，很有可能会花费大量时间以处理来自第三方库导致的类似问题。

规避第三方库的一个可行方法是*链接替换*。就WavReader应用而言，我们将创建一个包含存根函数的库，这些函数对应rlog中的每个函数，在构建应用可执行文件时，我们会链接到这个库。

链接替换，或链接期替换，听上去比做起来难得多，但其实可以很快建立。

8.9　为 rlog 创建测试替身

不需要为rlog库中定义的每个*函数*生成存根函数，只需定义测试需要的。我们开始注释掉makefile中的一些命令行，这个makefile将rlog和测试可执行文件链接起来。下面是对CMakeLists.txt

的更新（我们使用CMake）。

wav/7/CMakeLists.txt

```
project(SnippetPublisher)
cmake_minimum_required(VERSION 2.6)

include_directories($ENV{BOOST_ROOT}/ $ENV{RLOG_HOME} $ENV{CPPUTEST_HOME}/include)
link_directories($ENV{RLOG_HOME}/rlog/.libs $ENV{CPPUTEST_HOME}/lib)
set(Boost_USE_STATIC_LIBS ON)

add_definitions(-std=c++0x)

set(CMAKE_CXX_FLAGS "${CMAXE_CXX_FLAGS} -DRLOG_COMPONENT=debug -Wall")
set(sources WavReader.cpp WavDescriptor.cpp)
set(testSources WavReaderTest.cpp)
add_executable(utest testmain.cpp ${testSources} ${sources})
add_executable(SnippetPublisher main.cpp ${sources})

find_package(Boost $ENV{BOOST_VERSION} COMPONENTS filesystem system)
target_link_libraries(utest ${Boost_LIBRARIES})
target_link_libraries(utest CppUTest)
target_link_libraries(utest pthread)
target_link_libraries(utest rt)
➤ #target_link_libraries(utest rlog)

target_link_libraries(SnippetPublisher ${Boost_LIBRARIES})
target_link_libraries(SnippetPublisher pthread)
target_link_libraries(SnippetPublisher rlog)
```

在构建时，我们发现了许多链接错误。

```
Linking CXX executable utest
CMakeFiles/utest.dir/WavReader.cpp.o: In function `WavReader::WavReader(
   std::string const&, std::string const&)':
WavReader.cpp:(.text+0xef): undefined reference to
   `rlog::StdioNode::StdioNode(int, int)'
WavReader.cpp:(.text+0x158): undefined reference to
   `rlog::GetComponentChannel(char const*, char const*, rlog::LogLevel)'
WavReader.cpp:(.text+0x17c): undefined reference to
   `rlog::GetComponentChannel(char const*, char const*, rlog::LogLevel)'
WavReader.cpp:(.text+0x18e): undefined reference to
   `rlog::StdioNode::subscribeTo(rlog::RLogNode*)'
WavReader.cpp:(.text+0x1e4): undefined reference to
   `rlog::PublishLoc::~PublishLoc()'
...
```

需要为链接到rlog的每个函数提供存根。以下是一个方法。

(1) 将rlog的头文件复制进一个子目录，将其重命名为一个.cpp文件。

(2) 编辑这个.cpp文件，如果需要返回一个默认的返回值，那么就可以将函数原型变为一个存根了。

(3) 编译并重复上述两个步骤，直到消除所有的链接错误。

由于StdioNode出现在链接错误列表的第一个，我们就从它开始吧！

wav/8/StdioNode.cpp

```
#include <rlog/StdioNode.h>

class RLogNode;
class RLogData;

using namespace std;

namespace rlog {
   StdioNode::StdioNode(int _fdOut, int flags)
       : RLogNode() {}
   StdioNode::StdioNode(int _fdOut, bool colorizeIfTTY)
       : RLogNode(), fdOut( _fdOut ) { }
   StdioNode::~StdioNode() { }
   void StdioNode::subscribeTo( RLogNode *node ) { }
   void StdioNode::publish( const RLogData &data ) { }
}
```

这不是太难，但是将头文件转为实现文件可能会让人感到乏味，而且有时候有点棘手。（或许有一个很棒的工具能够帮助你做这些工作！）下面是一些需要考虑的事情（不是所有的事情）。

- 首先需要做的是，包含所创建的实现文件的地方头文件。
- 可能需要将实现文件放在一个命名空间中。
- 移除`virtual`和`static`关键字。
- 移除`public:`和其他一些访问修饰符。
- 移除成员变量和枚举值。
- 在移除预处理定义和typedefs时要小心。
- 如果函数有返回值，就返回一个最简单的，如0、false、""、空指针或一个无参构造函数构造的实例。
- 如果必须要返回一个`const`引用，创建一个全局变量，并返回这个变量。
- 不要忘记将函数限定在合适的类。
- 不要太注重外观，重要的是能够编译。

添加一个构建新存根库的makefile。

wav/8/rlog/CMakeLists.txt

```
project(rlogStub)
cmake_minimum_required(VERSION 2.6)

include_directories($ENV{RLOG_HOME})

add_definitions(-std=c++0x)

set(CMAKE_CXX_FLAGS "${CMAXE_CXX_FLAGS} -DRLOG_COMPONENT=debug -Wall")
set(sources StdioNode.cpp)
```

```
add_library(rlogStub ${sources})

target_link_libraries(rlogStub pthread)
```

接下来，更新测试构建脚本以便使用存根库。而产品应用SnippetPublisher继续使用产品级的rlog库。

wav/8/rlog/CMakeLists.txt

```
project(SnippetPublisher)
cmake_minimum_required(VERSION 2.6)

include_directories($ENV{BOOST_ROOT}/ $ENV{RLOG_HOME} $ENV{CPPUTEST_HOME}/include)
link_directories($ENV{RLOG_HOME}/rlog/.libs $ENV{CPPUTEST_HOME}/lib)
set(Boost_USE_STATIC_LIBS ON)

➤ add_subdirectory(rlog)
add_definitions(-std=c++0x)

set(CMAKE_CXX_FLAGS "${CMAXE_CXX_FLAGS} -DRLOG_COMPONENT=debug -Wall")
set(sources WavReader.cpp WavDescriptor.cpp)
set(testSources WavReaderTest.cpp)
add_executable(utest testmain.cpp ${testSources} ${sources})
add_executable(SnippetPublisher main.cpp ${sources})

find_package(Boost $ENV{BOOST_VERSION} COMPONENTS filesystem system)
target_link_libraries(utest ${Boost_LIBRARIES})
target_link_libraries(utest CppUTest)
target_link_libraries(utest pthread)
➤ target_link_libraries(utest rlogStub)

target_link_libraries(SnippetPublisher ${Boost_LIBRARIES})
target_link_libraries(SnippetPublisher pthread)
target_link_libraries(SnippetPublisher rlog)
```

构建基于`StdioNode.h`的存根失败了。我们为`RLogChannle.h`、`RLogNode.h`和`rlog.h`添加了存根，它们都是实现rlog库的头文件。下面是`RLogChannel.h`的存根实现，因为需要提供返回值，所以稍微有趣点。

wav/9/rlog/RLogChannel.cpp

```
#include "rlog/RLogChannel.h"
#include <string>
#include <iostream>

namespace rlog
{
RLogChannel::RLogChannel( const std::string &name, LogLevel level ){ }
RLogChannel::~RLogChannel(){}
void RLogChannel::publish(const RLogData &data){}

std::string nameReturn("");
```

```
const std::string &RLogChannel::name() const { return nameReturn; }

LogLevel RLogChannel::logLevel() const { return LogLevel(); }
void RLogChannel::setLogLevel(LogLevel level) {}
RLogChannel *getComponent(RLogChannel *componentParent,
                    const char *component){ return 0; }
}
```

我们一次只创建一个存根，同时记住，要更新每个makefile文件。

wav/9/rlog/CMakeLists.txt

```
set(sources rlog.cpp RLogChannel.cpp RLogNode.cpp StdioNode.cpp)
```

尝试构建已有的新存根。在准备好4个存根后，链接和测试成功了！大概需要20分钟。花费这点精力消除此类烦人的事情还是非常值得的。

8.10 测试驱动开发改动

现在可以将支持多通道的代码变动测试驱动开发进`writeData()`函数了。通道数表示了同时音轨（来自于不同声源的声音）的数目，这些音轨的声音是同时播放的。对于单声道输出，通道数为1。对于立体音输出，通道数为2。播放一个WAV文件需要按顺序遍历所有的采样点。一个采样由一系列子采样构成，每个通道一个。具有4个采样的音频片段的每个采样是一个字节，如下：

```
AA BB CC DD
```

假设第二个通道的采样顺序如下：

```
01 02 03 04
```

那么最终的WAV流应该和下面一样：

```
AA 01 BB 02 CC 03 DD 04
```

下面这个测试展示了通道数目的增加是怎样在一个数据流占据更多字节的。

wav/10/WavReaderTest.cpp

```
#include "CppUTest/TestHarness.h"
#include "WavReader.h"
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

TEST_GROUP(WavReader) {
};

TEST(WavReader_WriteSamples, IncorporatesChannelCount) {
   char data[] { "0123456789ABCDEFG" };
```

```
    uint32_t bytesPerSample { 2 };
    uint32_t startingSample { 0 };
    uint32_t samplesToWrite { 2 };
    uint32_t channels { 2 };
    reader.writeSamples(
          &out, data, startingSample, samplesToWrite, bytesPerSample, channels);
    CHECK_EQUAL("01234567", out.str());
 }
```

为了避免现在就去改动其他测试，可以使用默认的channels参数。和往常一样，我们的目的是先让测试通过，然后再整理代码。

wav/10/WavReader.h

```
  void writeSamples(std::ostream* out, char* data,
        uint32_t startingSample,
        uint32_t samplesToWrite,
        uint32_t bytesPerSample,
➤       uint32_t channels=1);
```

现在我们实现让新测试通过的代码。

wav/10/WavReader.cpp

```
  void WavReader::writeSamples(ostream* out, char* data,
        uint32_t startingSample,
        uint32_t samplesToWrite,
        uint32_t bytesPerSample,
➤       uint32_t channels) {
     rLog(channel, "writing %i samples", samplesToWrite);

     for (auto sample = startingSample;
          sample < startingSample + samplesToWrite;
          sample++) {
➤       auto byteOffsetForSample = sample * bytesPerSample * channels;
➤
➤       for (uint32_t channel{0}; channel < channels; channel++) {
➤          auto byteOffsetForChannel =
➤             byteOffsetForSample + (channel * bytesPerSample);
➤          for (uint32_t byte{0}; byte < bytesPerSample; byte++)
➤             out->put(data[byteOffsetForChannel + byte]);
➤       }
     }
  }
```

还有一些其他事情需要修正。当写出新的长度为10秒的WAV文件时，我们也更新了数据段的长度。但由于没有改对通道数，现在的长度是错的。虽然只是一行代码，但我们还是乐于将其提取为一个函数，以便测试和改正。遵从相同的方法吧！写一个测试以刻画已有的行为，修改测试来定义新的行为，并修改产品代码。以下是改完后的测试、提取得来的函数和open()中的客户端代码，它调用了这个新的函数。

wav/11/WavReaderTest.cpp

```
TEST_GROUP(WavReader_DataLength) {
   WavReader reader{"",""};
};

TEST(WavReader_DataLength, IsProductOfChannels_BytesPerSample_and_Samples) {
   uint32_t bytesPerSample{ 2 };
   uint32_t samples { 5 };
   uint32_t channels { 4 };

   uint32_t length { reader.dataLength(bytesPerSample, samples, channels) };

   CHECK_EQUAL(2 * 5 * 4, length);
}
```

wav/11/WavReader.cpp

```
   // ...
   rLog(channel, "total seconds %u ", totalSeconds);

➤  dataChunk.length = dataLength(
➤        samplesToWrite,
➤        bytesPerSample,
➤        formatSubchunk.channels);

   out.write(reinterpret_cast<char*>(&dataChunk), sizeof(DataChunk));
   // ...

uint32_t WavReader::dataLength(
      uint32_t samples,
      uint32_t bytesPerSample,
      uint32_t channels
      ) const {
   return samples * bytesPerSample * channels;
}
```

8.11 新的场景

场景：增强描述符

背景：open()函数的倒数第二步是发送消息至WavDescriptor对象，该对象的工作是将格式化的记录追加至描述符文件。WAV发布商的用户界面用这个描述符文件中的内容来显示可用的WAV文件。

这个描述符接收WAV文件名、（剪辑前的）总时间、每秒的采样数以及通道的数目。

场景：作为内容转销商，我想要WAV文件中的描述符也包括新的剪辑文件长度。修改描述符对象使其接受文件长度。

会有其他开发者改变WavDescriptor的实现，使每个记录包含剪辑文件长度（但愿他们在使用TDD！）。我们的工作只是改变WavReader的实现。

完成此事需要做两件事。第一，计算或取得文件大小；第二，证明我们将此文件大小值传给了WavDescriptor。

加入新的功能时，最好先将其作为一个新的方法，甚至一个新的类。这将确保代码的测试覆盖率不会因为代码库的增长而降低。同时，也有助于恪守单一责任原则，并且从小方法和类中受益。

我们需要一个函数，可以在给定一个文件名时，就返回其大小。为了测试驱动开发这个新功能，我们这样做：

wav/12/FileUtilTest.cpp

```
// 耗时的测试，因为需要和文件系统交互
TEST_GROUP_BASE(FileUtil_Size, FileUtilTests) {
};

TEST(FileUtil_Size, AnswersFileContentSize) {
   string content("12345");
   createTempFile(content);

   size_t expectedSize { content.length() + sizeof('\0') };
   LONGS_EQUAL(expectedSize, (unsigned)util.size(TempFileName));
}
```

wav/12/FileUtil.h

```
class FileUtil {
public:
   std::streamsize size(const std::string& name) {
      std::ifstream stream{name, std::ios::in | std::ios::binary};
      stream.seekg(0, std::ios::end);
      return stream.tellg();
   }
};
```

这个测试有什么问题吗？它不够快（参见4.3节）。

在TDD过程中，你有可能最终只能得到少数几个运行慢的测试。或许这没什么问题，但要努力使这样的测试减少，尽量降至零。更重要的是，标出这些慢速的测试，并确保能够找到一种运行快速、慢速，或者同时运行快速和慢速测试集的方法。

8.12　寻求更快测试的简要探索

目前，对WAV Snippet Publiser代码库来说，只有一个慢速测试。我们将努力保持这样。但如果需要另一个基于文件的实用程序，要确保不再添加第二个慢速测试。一旦放任不管，慢速测试数量会很快多起来。

一个可行的方案是将部分只处理音频流的功能独立出来，并测试驱动开发它。对于`size()`函数，我们甚至可以创建更小的方法。

wav/12/StreamUtilTest.cpp

```
TEST(StreamUtil_Size, AnswersNumberOfBytesWrittenToStream) {
   istringstream readFrom{"abcdefg"};

   CHECK_EQUAL(7, StreamUtil::size(readFrom));
}
```

wav/12/StreamUtil.cpp

```
std::streamsize size(std::istream& stream) {
   stream.seekg(0, std::ios::end);
   return stream.tellg();
}
```

FileUtil类的`size()`函数只是简单调用StreamUtil类的`size()`函数，并传递了一个`ifstream`引用。我们可以认为这个文件实用函数很小，以至于不会被破坏，不需要为之编写测试。

我们也可以创建FileUtil类成员函数`execute()`，这个函数唯一要做的就是创建一个ifstream对象，然后将它传递给操作此对象的函数。客户端代码可能会给`execute()`传递一个函数指针。

wav/12/FileUtilTest.cpp

```
streamsize size = util.execute(TempFileName,
              [&](istream& s) { return StreamUtil::size(s); });
```

wav/12/FileUtil.h

```
std::streamsize execute(
      const std::string& name,
      std::function<std::streamsize (std::istream&)> func) {
   std::ifstream stream{name, std::ios::in | std::ios::binary};
   return func(stream);
}
```

这样做的好处是，我们只需要为`execute()`写一个测试来和文件打交道。其他测试仅仅基于流，执行会很快。

8.13　立竿见影的提取

我们需要找出`open()`函数中哪些地方调用了`size()`。这个调用可能会在`open()`函数结尾，在给`descriptor`对象发送消息之前。但因为`size()`会重新打开文件，所以需要确保新的WAV文件首先被关闭。

不幸的是，理解`open()`函数体的结构具有挑战性。调用descriptor成员函数之前的代码充斥着文件的读和写。编写一个测试执行整个函数依然相当困难。（可以给`open()`传入一个真实、经过验证的WAV文件，但这样就会得到一个慢速、对外有依赖的测试。）

相反，我们重构`open()`，目的是得出一些可以存根或模拟的函数。经过几十分钟的函数提取，代码看上去好多了。虽然有些地方的代码仍然有点丑，但得到的函数更容易让人理解消化了。

wav/13/WavReader.cpp

```
void WavReader::open(const std::string& name, bool trace) {
   rLog(channel, "opening %s", name.c_str());

   ifstream file{name, ios::in | ios::binary};
   if (!file.is_open()) {
      rLog(channel, "unable to read %s", name.c_str());
      return;
   }

   ofstream out{dest_ + "/" + name, ios::out | ios::binary};

   FormatSubchunk formatSubchunk;
   FormatSubchunkHeader formatSubchunkHeader;
   readAndWriteHeaders(name, file, out, formatSubchunk, formatSubchunkHeader);

   DataChunk dataChunk;
   read(file, dataChunk);

   rLog(channel, "riff header size = %i" , sizeof(RiffHeader));
   rLog(channel, "subchunk header size = %i", sizeof(FormatSubchunkHeader));
   rLog(channel, "subchunk size = %i", formatSubchunkHeader.subchunkSize);
   rLog(channel, "data length = %i", dataChunk.length);

   auto data = readData(file, dataChunk.length); // 泄漏!

   writeSnippet(name, file, out, formatSubchunk, dataChunk, data);
}

void WavReader::read(istream& file, DataChunk& dataChunk) {
   file.read(reinterpret_cast<char*>(&dataChunk), sizeof(DataChunk));
}

char* WavReader::readData(istream& file, int32_t length) {
   auto data = new char[length];
   file.read(data, length);
   //file.close(); // istreams 是 RAII
   return data;
}

void WavReader::readAndWriteHeaders(
      const std::string& name,
      istream& file,
      ostream& out,
      FormatSubchunk& formatSubchunk,
      FormatSubchunkHeader& formatSubchunkHeader) {
   RiffHeader header;
   file.read(reinterpret_cast<char*>(&header), sizeof(RiffHeader));
   // ...
}

void WavReader::writeSnippet(
      const string& name, istream& file, ostream& out,
      FormatSubchunk& formatSubchunk,
```

```
    DataChunk& dataChunk,
    char* data
    ) {
uint32_t secondsDesired{10};
if (formatSubchunk.bitsPerSample == 0) formatSubchunk.bitsPerSample = 8;
uint32_t bytesPerSample{formatSubchunk.bitsPerSample / uint32_t{8}};

uint32_t samplesToWrite{secondsDesired * formatSubchunk.samplesPerSecond};
uint32_t totalSamples{dataChunk.length / bytesPerSample};

samplesToWrite = min(samplesToWrite, totalSamples);

uint32_t totalSeconds{totalSamples / formatSubchunk.samplesPerSecond};

rLog(channel, "total seconds %u ", totalSeconds);

dataChunk.length = dataLength(
        samplesToWrite,
        bytesPerSample,
        formatSubchunk.channels);
out.write(reinterpret_cast<char*>(&dataChunk), sizeof(DataChunk));

uint32_t startingSample{
    totalSeconds >= 10 ? 10 * formatSubchunk.samplesPerSecond : 0};

writeSamples(&out, data, startingSample, samplesToWrite, bytesPerSample);

rLog(channel, "completed writing %s", name.c_str());

descriptor_->add(dest_, name,
        totalSeconds, formatSubchunk.samplesPerSecond, formatSubchunk.channels);

//out.close(); // ostreams是RAI
```

因为现在`writeSnippet()`接收一个输入流和一个输出流，所以它就不再直接依赖文件系统。现在为其写一个测试或者为`read()`和`readData()`写测试似乎合理了。在写测试前，还要将`readAndWriteHeaders()`（没有完全列出来）重构为更易于管理的代码，这样做不会花费太长时间。

我们甚至还发现了一个可能的内存泄漏问题。将代码碎化成更小的函数使缺陷变得显而易见。

在重构过程中，我们删除to-do并注释掉关闭流的调用。我们需要马上修正选择，但也没必要显式地关闭文件（std::ofstream支持RAII[①]），`close()`也不是std::ostream的接口。或许我们的分析已经够多了，现在是时候手动或自动地运行手头的测试了。

我们做好处理文件大小场景的准备了吗？多一点重构或许会使事情变得更简单，既然`writeSnippet()`已经是小且专注的函数，让我们研究一下能怎么测试它。我们将会写一些测试来刻画其各类行为。

① RAII即Resource Acquisition Is Intialization，这是一个编程惯用法，起源于编写异常安全的C++代码。参见https://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization。——译者注

8.14 用成员变量查看状态

首先要确保正确地计算出`totalSeconds`，这个值会传给descriptor。（如果不准备改动`writeSnippet()`函数，或许就不需要为之写测试。这里写了一个测试来演示特殊的技巧。）

wav/14/WaveReaderTest.cpp

```
TEST_GROUP(WavReader_WriteSnippet) {
   WavReader reader{"",""};
   istringstream input{""};
   FormatSubchunk formatSubchunk;
   ostringstream output;
   DataChunk dataChunk;
   char* data;
   uint32_t TwoBytesWorthOfBits{2 * 8};

   void setup() override {
      data = new char[4];
   }

   void teardown() override {
      delete[] data;
   }
};

TEST(WavReader_WriteSnippet, UpdatesTotalSeconds) {
   dataChunk.length = 8;
   formatSubchunk.bitsPerSample = TwoBytesWorthOfBits;
   formatSubchunk.samplesPerSecond = 1;

   reader.writeSnippet("any", input, output, formatSubchunk, dataChunk, data);

   CHECK_EQUAL(8 / 2 / 1, reader.totalSeconds);
}
```

在这个测试中，我们做了大量的初始化工作。因为这些工作和测试最终期待的结果不相干，所以可以将这些初始化代码放在WavReader_WriteSnippet测试组（fixture）定义中。 同时，也可把WavReader.cpp中的`struct`定义移至WavReader.h中，这样测试就能访问这些结构了。

等等！如果`totalSeconds`是`writeSnippet()`的局部变量，那怎么去验证它的值呢？答案很简单：`totalSeconds`现在是公共的成员变量！

wav/14/WavReader.h

```
public:
   // ...
   uint32_t totalSeconds;
```

wav/14/WavReader.cpp

```
void WavReader::writeSnippet(
      const string& name, istream& file, ostream& out,
```

```
      FormatSubchunk& formatSubchunk,
      DataChunk& dataChunk,
      char* data
      ) {
   uint32_t secondsDesired{10};
   if (formatSubchunk.bitsPerSample == 0) formatSubchunk.bitsPerSample = 8;
   uint32_t bytesPerSample{formatSubchunk.bitsPerSample / uint32_t{8}};

   uint32_t samplesToWrite{secondsDesired * formatSubchunk.samplesPerSecond};
   uint32_t totalSamples{dataChunk.length / bytesPerSample};

   samplesToWrite = min(samplesToWrite, totalSamples);

➤  totalSeconds = totalSamples / formatSubchunk.samplesPerSecond;

   rLog(channel, "total seconds %u ", totalSeconds);
   // ...
}
```

测试偷偷地查看本应对外不可见的totalSeconds。万恶之源啊！有时为了能够把控遗留代码，我们需要做些弄脏代码的事情。要提醒自己进一步改动代码，若不知道是否会破坏一些功能，则会把代码弄得更脏。

除此之外，还有一个应对这种情况的更好方法。

8.15 用 mock 查看状态

writeSnippet()函数的目的之一是将总秒数传给descriptor。在前一节中，我们将其改为成员变量以便查看此值。我们也可以让WavReader用descriptor的测试替身来得到发送给descriptor的值。

你已经在第5章中学习了怎样用Google Mock创建测试替身。由于本章示例中使用的是CppUTest，我们将使用其自带的mock工具——CppUMock。和Google Mock一样，我们定义了WavDescriptor的派生类，它将查看发送到其add()函数的消息。

wav/15/WavReaderTest.cpp

```
class MockWavDescriptor : public WavDescriptor {
public:
   MockWavDescriptor(): WavDescriptor("") {}
   void add(
      const string&, const string&,
      uint32_t totalSeconds,
      uint32_t, uint32_t) override {
➤     mock().actualCall("add")
➤        .withParameter("totalSeconds", (int)totalSeconds);
   }
};
```

为了覆写add()，我们需要将其在WavDescriptor中的声明改为虚拟函数。

wav/15/WavDescriptor.h

```
➤ virtual void add(
➤    const std::string& dir, const std::string& filename,
➤    uint32_t totalSeconds, uint32_t samplesPerSecond,
➤    uint32_t channels) {
➤    // ...
     WavDescriptorRecord rec;
     cpy(rec.filename, filename.c_str());
     rec.seconds = totalSeconds;
     rec.samplesPerSecond = samplesPerSecond;
     rec.channels = channels;

     outstr->write(reinterpret_cast<char*>(&rec), sizeof(WavDescriptorRecord));
  }
```

MockWavDescriptor中标记出来的代码行命令CppUTest全局对象MockSupport（通过调用`mock()`获得）记录对名为`add`的函数的调用。MockSupport对象会同时得到参数名为`totalSeconds`的值。（我用括号将这些名称括起来是因为在使用CppUMock时可以任意命名。这是一个相对方便的反射。）

通过WavReader的构造函数，我们将测试替身注入WavReader对象中。

wav/15/WavReader.cpp

```
  TEST_GROUP(WavReader_WriteSnippet) {
➤    shared_ptr<MockWavDescriptor> descriptor{new MockWavDescriptor};
➤    WavReader reader{"", "", descriptor};
     istringstream input{""};
     FormatSubchunk formatSubchunk;
     ostringstream output;
     DataChunk dataChunk;
     char* data;
     uint32_t TwoBytesWorthOfBits{2 * 8};
     void setup() override {
        data = new char[4];
     }

     void teardown() override {
        mock().clear();
        delete[] data;
     }
  };
```

在测试中，我们会通知MockSupport对象期望一个名为`add`的函数被调用。同时通知它，调用的`totalSeconds`的参数为一个特定的值。测试的这种安排就是设立期望。一旦`writeSnippet()`被调用，测试的断言部分就会验证加入到MockSupport对象中的所有期望是否满足。

wav/15/WavReaderTest.cpp

```
TEST(WavReader_WriteSnippet, UpdatesTotalSeconds) {
   dataChunk.length = 8;
   formatSubchunk.bitsPerSample = TwoBytesWorthOfBits;
```

```
   formatSubchunk.samplesPerSecond = 1;
➤  mock().expectOneCall("add").withParameter("totalSeconds", 8 / 2 / 1);
   reader.writeSnippet("any", input, output, formatSubchunk, dataChunk, data);
➤  mock().checkExpectations();
}
```

相应地，我们将descriptor指针改为共享指针。使用共享指针允许测试和产品代码正确地管理descriptor对象的创建与删除。同时，为了最小化对现有测试的影响，我们将descriptor的构造参数默认为空指针。

wav/15/WavReader.h

```
class WavReader {
public:
   WavReader(
         const std::string& source,
         const std::string& dest,
         std::shared_ptr<WavDescriptor> descriptor=0);
   // ...
private:
   // ...
   std::shared_ptr<WavDescriptor> descriptor_;
}
```

wav/15/WavReader.cpp

```
WavReader::WavReader(
      const std::string& source,
      const std::string& dest,
➤     shared_ptr<WavDescriptor> descriptor)
   : source_(source)
   , dest_(dest)
➤  , descriptor_(descriptor) {
➤  if (!descriptor_)
➤     descriptor_ = make_shared<WavDescriptor>(dest);

   channel = DEF_CHANNEL("info/wav", Log_Debug);
   log.subscribeTo((RLogNode*)RLOG_CHANNEL("info/wav"));

   rLog(channel, "reading from %s writing to %s", source.c_str(), dest.c_str());
}

WavReader::~WavReader() {
➤  descriptor_.reset();
   delete channel;
}
```

不需要改变正在测试的`writeSnippet()`函数中的任何代码！`writeSnippet()`依旧对descriptor调用`add()`，而不需要知道descriptor是产品代码WavDescriptor的类实例还是测试替身。

通过测试驱动开发`writeSnippet()`来获取和传递文件大小，我们终于可以完成场景中所描述的需求了。实际上，我们选择测试UpdatesTotalSecond，并更新它来验证这两个参数。为了支

持给定获取文件大小请求的存根回馈，我们创建了FileUtil的mock。FileUtil的mock是通过setter函数而非构造函数注入的。

wav/16/WavReaderTest.cpp

```
class MockWavDescriptor : public WavDescriptor {
public:
   MockWavDescriptor(): WavDescriptor("") {}
   void add(
      const string&, const string&,
      uint32_t totalSeconds,
      uint32_t, uint32_t,
      uint32_t fileSize) override {
      mock().actualCall("add")
         .withParameter("totalSeconds", (int)totalSeconds)
➤        .withParameter("fileSize", (int)fileSize);
   }
};

➤ class MockFileUtil: public FileUtil {
➤ public:
➤    streamsize size(const string& name) override {
➤       return mock().actualCall("size").returnValue().getIntValue();
➤    }
➤ };

TEST_GROUP(WavReader_WriteSnippet) {
   shared_ptr<MockWavDescriptor> descriptor{new MockWavDescriptor};
   WavReader reader{"", "", descriptor};

➤  shared_ptr<MockFileUtil> fileUtil{make_shared<MockFileUtil>()};

   istringstream input{""};
   FormatSubchunk formatSubchunk;
   ostringstream output;
   DataChunk dataChunk;
   char* data;
   uint32_t TwoBytesWorthOfBits{2 * 8};

   const int ArbitraryFileSize{5};

   void setup() override {
      data = new char[4];
➤     reader.useFileUtil(fileUtil);
   }

   void teardown() override {
      mock().clear();
      delete[] data;
   }
};

TEST(WavReader_WriteSnippet, SendsFileLengthAndTotalSecondsToDescriptor) {
```

```
   dataChunk.length = 8;
   formatSubchunk.bitsPerSample = TwoBytesWorthOfBits;
   formatSubchunk.samplesPerSecond = 1;

➤  mock().expectOneCall("size").andReturnValue(ArbitraryFileSize);

   mock().expectOneCall("add")
      .withParameter("totalSeconds", 8 / 2 / 1)

➤     .withParameter("fileSize", ArbitraryFileSize);

   reader.writeSnippet("any", input, output, formatSubchunk, dataChunk, data);

   mock().checkExpectations();
}
```

wav/16/WavReader.cpp

```
void WavReader::writeSnippet(
      const string& name, istream& file, ostream& out,
      FormatSubchunk& formatSubchunk,
      DataChunk& dataChunk,
      char* data
      ) {
   // ...
   writeSamples(&out, data, startingSample, samplesToWrite, bytesPerSample);

   rLog(channel, "completed writing %s", name.c_str());

➤  auto fileSize = fileUtil_->size(name);

   descriptor_->add(dest_, name,
        totalSeconds, formatSubchunk.samplesPerSecond, formatSubchunk.channels,
➤       fileSize);

   //out.close(); // ostreams 是 RAII
}
```


8.16 其他注入技巧

到目前为止，我们介绍了很多创新性的技巧，这些技巧可以提升应对难以测试的代码的能力。我们使用构造函数注入、setter函数注入和成员变量探测。你在第5章中学到的技巧，如覆写工厂方法或通过模板参数，都可以在这里派上用场。如果真的需要，你甚至可以尝试更富创新性的技巧，例如，使用预处理器重新定义代码。

避免只使用一种打破依赖的技巧而陷入钉锤陷阱。《修改代码的艺术》介绍了许多有用的模式。你要做的就是熟悉这些模式，以便选择最适合当前处境的方法。

8.17　用 Mikado 方法大规模改动代码

有时你需要大规模地改动代码。代码或许会传递分散的变量，这时你会想将它们聚合起来放进一个结构体中，又或者你想要重新组织被许多客户使用的大型类。可能需要几天或几个星期来完成这些改动。这几天或几周的日子会很难熬，因为你必须努力改动代码。

在开始为改动做扫尾工作时，通常你会发现更多需要改动的代码。你或许会发现，这些要改动的代码自身会牵扯出与当前改动无关的、但又必须先完成的改动。于是，几天就会变成几周或更长时间。

你甚至可能没有时间来做这些工作。公司业务发展可能会提高另一个任务的优先级，迫使你放弃提升当前代码的工作。假如你的工作在主线（trunk）上，那么就有可能因为放弃未完成的解决方案而使代码处于更差的状态。更糟糕的是，即便到目前为止一直很努力，你或许还是没能弄清楚当前的进展和那些未完成的工作。同时，只完成一半的解决方案会妨碍其他人的改动工作，这会使团队的其他成员感到疑惑和沮丧。即使最终重拾最初的重构工作，也可能要花大量时间回想当时的进度，以及还有哪些工作没有做。

如果起初是在分支上做出大量的改动，那么你会发现，要合并的代码变多了，且需要更长的时间来完成合并。如果其他开发者修改的代码和你修改的代码有交集（显然，他们没有为你的代码健康考虑），那么即使增量地合并也会令你沮丧透顶。“可以等我完成吗？”不。

由Daniel Brolund和Ola Ellnestam发明的Mikado方法提供了一个有章可循的方法。你可以在*The Mikado Method* [EB14]中深入阅读关于此技巧的知识。本章余下的部分将简要概览此过程，并带你一同体验此技巧的使用。

8.18　Mikado 方法概览

完整的过程要经历看上去令人却步的9个步骤。然而，一旦经历过几次后，你会感到这再自然不过了。习惯该方法甚至比习惯TDD步骤还要快。

(1) 制定Mikado目标以及想要达到的最终状态。
(2) 以最稚拙的方法实现这一目标。
(3) 找出所有的错误。
(4) 找到解决错误的即时方案。
(5) 将即时方案作为新的前提目标。
(6) 如果有错误，将相应的代码回滚到最初状态。
(7) 对每个方案重复步骤2至步骤6。
(8) 检查代码是否存在错误；将前提目标标记为完成。
(9) 完成了Mikado目标吗？如果是，任务完成！

做大规模重构工作的大部分时间最终并不是花在实际的代码改动本身。通常来说，每个改动都很小（虽然改动很多）。相反，大部分时间花在必要条件分析和自我思索上。“调用这个函数的所有客户代码在哪？如果改动它们，还有哪些客户代码会受到影响？代码到底做了什么？我要怎样证明代码依然工作？”

你可以用Mikado方法将分析和基于分析改动代码的时间相分离。可以将代码改动约简为简单的改动图，或Mikado图，即一些脚本。这个图用于描绘主目标，即Mikado目标，以及主目标所依赖的一系列其他目标。通常而言，将图中改动的元素依次应用到代码库中。这个动作能让你以高度的自信快速、正确地改动代码。

通过建立Mikado图，你可以对必须要完成的任务形成可视化的总结，还可以将这些任务分发给团队成员以加速达成Mikado目标。Mikado图为团队间的沟通和协作提供了一个中心点。

8.19 用 Mikado 移动一个方法

我们可以将WAV片段或WAV片段写出器（snippet writer）作为独立的抽象。主类WavReader的名字已经表明，此类中的任何写出片段都不是其职责。此外，我们正在大量地修改片段逻辑，并想将这些改动局限于更小的类范围。

作为短期的Mikado目标，我们要将`writeSnippet()`提取为一个类，即Snippet。一旦完成这个即时目标，我们便计划相应地整理Snippet代码。但是，每次只集中一个目标。

考虑到篇幅限制，这个示例相对较小，但其真实性与日常做的重构一样。即使这是个很小的任务，你还是会惊讶于即将经历的许多步骤。Mikado方法让你不会在穿梭于这些步骤时丢下或完全漏掉目标。

作为Mikado方法的第一步，我们用同心的双椭圆表示最终目标。我们倾向于将一个大的白板作为工具。因为我们希望每个人都能看到Mikado图，所以需要大量的空间，在修改代码的过程中，我们会增量地构建这个图。



根据Mikado方法的第二步，我们以最稚拙的方法直奔目标。将`writeSnippet()`函数移至（剪切并粘贴）一个新文件Snippet.h中，并将其放入一个类定义中。

wav/17/Snippet.h

```
#ifndef Snippet_h
#define Snippet_h
```

```
class Snippet {
public:
   void writeSnippet(
         const string& name, istream& file, ostream& out,
         FormatSubchunk& formatSubchunk,
         DataChunk& dataChunk,
         char* data
         ) {
      // ...
   uint32_t secondsDesired{10};
   if (formatSubchunk.bitsPerSample == 0) formatSubchunk.bitsPerSample = 8;
   uint32_t bytesPerSample{formatSubchunk.bitsPerSample / uint32_t{8}};

   uint32_t samplesToWrite{secondsDesired * formatSubchunk.samplesPerSecond};
   uint32_t totalSamples{dataChunk.length / bytesPerSample};

   samplesToWrite = min(samplesToWrite, totalSamples);

   uint32_t totalSeconds { totalSamples / formatSubchunk.samplesPerSecond };

   rLog(channel, "total seconds %i ", totalSeconds);

   dataChunk.length = dataLength(
         samplesToWrite,
         bytesPerSample,
         formatSubchunk.channels);
   out.write(reinterpret_cast<char*>(&dataChunk), sizeof(DataChunk));

   uint32_t startingSample{
      totalSeconds >= 10 ? 10 * formatSubchunk.samplesPerSecond : 0};

   writeSamples(&out, data, startingSample, samplesToWrite, bytesPerSample);

   rLog(channel, "completed writing %s", name.c_str());

   auto fileSize = fileUtil_->size(name);
   descriptor_->add(dest_, name,
         totalSeconds, formatSubchunk.samplesPerSecond, formatSubchunk.channels,
         fileSize);
   //out.close(); // ostreams是RAII
   }
};

#endif
```

我们在WavReader中创建一个Snippet对象，然后调用其`writeSnippet()`成员函数。

wav/17/WavReader.cpp

```
#include "Snippet.h"
// ...
void WavReader::open(const std::string& name, bool trace) {
   // ...
   auto data = readData(file, dataChunk.length); // 泄漏!
```

```
➤    Snippet snippet;
➤    snippet.writeSnippet(name, file, out, formatSubchunk, dataChunk, data);
  }
```

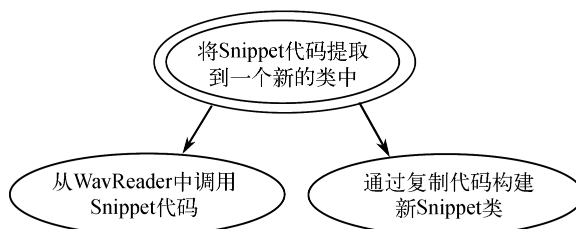
进行编译（Mikado方法的第三步：找出所有的错误）。快速检查编译错误发现，因为在Snippet.h中漏掉了一些定义，所以调用`writeSnippet()`的代码无法编译。

为了完成Mikado目标，我们需要完成两个前提任务（Mikado方法的第四步：找到解决错误的即时方案）：

(1) 粘贴代码以便得到一个新Snippet类；

(2) 修改WavReader以便使用Snippet对象（我们已经尝试了修改代码）。

将此改进作为前提更新进Mikado图（Mikado方法的第五步：将即时方案作为新的前提目标）。



稍后，我们会回头处理从WavReader中调用Snippet的代码。现在先集中创建Snippet类，它是一个可以持续丰富的独立类。

最有趣的部分是Mikado方法的第六步。因为有错误，所以我们回滚代码改动。是的，丢弃修改的代码。我们已经得到了分析结果，并标记在了Mikado图中，一旦知道要做什么，真正改动代码不会花费太长时间（我们会在后文中讨论到）。

如果你在使用像git一样好的源代码管理工具，那么Mikado方法的第六步会非常简单：

```
git reset —hard && git clean -f
```

这当然可能会导致数据丢失。< Insert standard legal disclaimer here.>

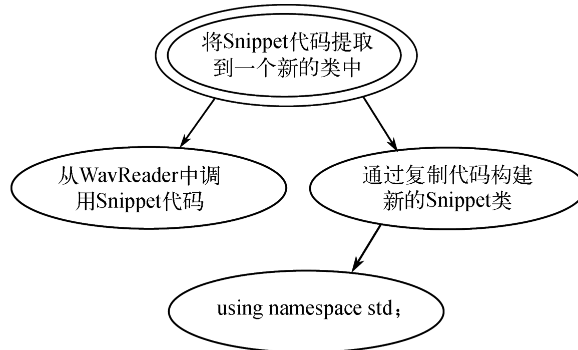
继续！我们的目标是，不断地走向Mikado图的叶节点——不需要任何前提动作就可以完成的任务，不断为每个节点重复前面所述的Mikado步骤。现在要做的是，判断通过复制代码而创建新的Snippet类是否为叶节点。第一次尝试基本和之前写的代码一样：将`writeSnippet()`复制到Snippet.h，把它放进类定义中，从声明中移除`WavReader::`。为了让编译器找到Snippet.h，在WavReader中加入`#include`。

对一些人来说，选择可能会失败的简单目标有点令人厌烦。但这使我们不必在没有任何具体反馈前就没完没了地做分析。而且，这也有助于在Mikado图中勾勒更小的子步骤。

我们再次收到了关于未识别std类型的编译错误，但这次的编译器输出没有和WavReader中的问题混在一起。为了加速达成Mikado目标，我们决定加入`using`指示符来解决问题。是的，这“不

好”。我们先做个记录，等完成Mikado目标后再整理代码。（也可以将整理代码作为Mikado图的一个节点，但我们目前先保持示例简单化。）

解决错误的方案生成了新的Mikado目标。



Mikado图有点简略，也许不能满足你的要求，但可以满足我们的需求。我们的目标包括命名空间声明、Snippet.h和类定义。

你或许在思索Mikado方法中每一步的粒度。是否应该详细地记录代码操作呢？答案取决于问题的难易程度、你是否希望获得别人的帮助，以及加入到任务中的人理解到的隐含之意有哪些。对大多数任务而言，你可以总结详细的步骤，假设会记住所需的细节，或者你的搭档知道要做什么。如果想要其他人通过阅读Mikado图就可以改动代码，你或许还需要包含更多的信息。

我们的Mikado图展示了一个相对低层次的目标。对于更大的Mikado目标而言，我们要将整个目标（将Snippet代码提取成新的类）作为单独的节点。

因为包含`using`指示符来修复编译错误失败了，所以我们再一次回滚代码。或许你仍然对回滚代码的理念有点反感，但记住，这有助于构建增量的、文档化很好的解决方案路径。

回滚代码会变得非常自然。重新改动代码花费的时间会大幅减少。你会偶尔尝试简单的方案，但基本行不通。回滚和重来带来的心理舒适感使你觉得压力全无。

现在从“using namespace std;”目标开始，并尝试解决它。在WavReader中加入`#include`语句。然后创建Snippet.h，向其中加入一个类，并加入`using`指示符。

wav/18/Snippet.h

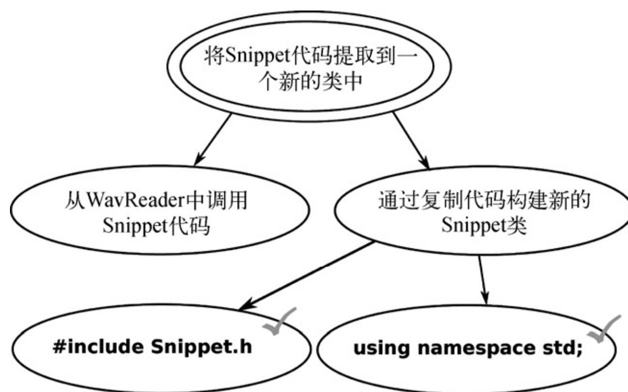
```
#ifndef Snippet_h
#define Snippet_h

using namespace std;

class Snippet {
public:
};
```

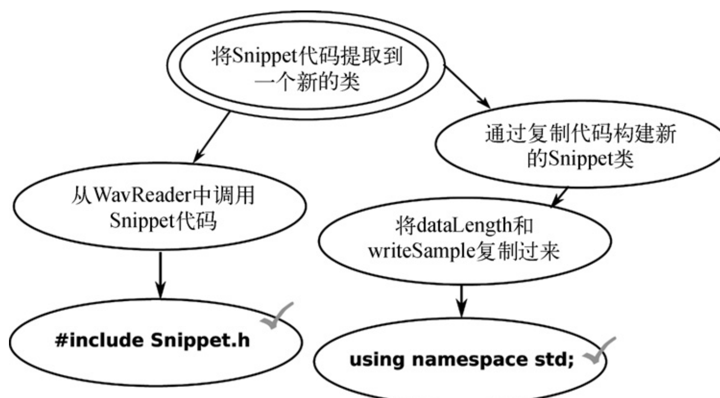
```
#endif
```

可以了！测试全部通过了。我们已经完成了“using namespace std;”的目标，可以在不破坏任何功能的前提下加入到代码库了。这时我们可以决定是否要提交代码。因为使用了强大的源代码版本工具，所以我们可以经常提交代码，这是安全的，也不要介意代码改动有多小。（如果有很多代码提交，使用git可以将它们合并为一个。）



我们将这个目标从Mikado图中勾掉。同时，将加入WavReader中的`#include`语句标记为已完成的目标。

我们做了另外一个尝试来实现“通过复制代码构建新的Snippet类”的目标。现在的错误更少了，这说明有几个成员（既有函数，也有变量）未被定义。现在来看函数`dataLength()`和`writeSamples()`，我们发现，它们不依赖任何成员变量。将复制这两个函数作为下一个前提目标最合适不过了。



再一次回滚代码并复制函数……哦……失败了！原来有一个成员变量`channels`。（不仅该变量没有帮助识别成员变量的下划线尾缀，而且内嵌的循环重用了这个变量。呃，将这个加入稍后

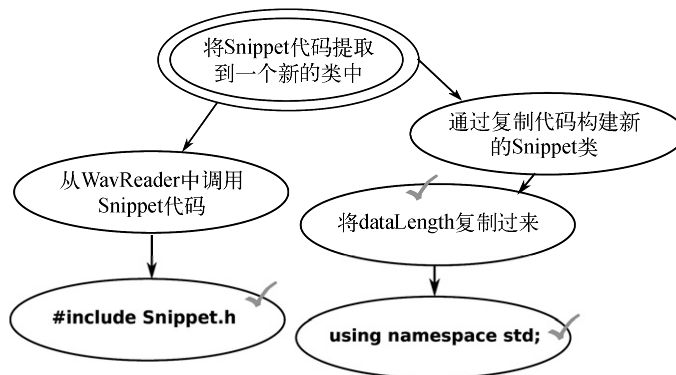
修复的列表中。）

先回滚代码，并将当前目标缩减至只复制dataLength()。

wav/19/Snippet.h

```
class Snippet {
public:
➤   uint32_t dataLength(
➤         uint32_t samples,
➤         uint32_t bytesPerSample,
➤         uint32_t channels
➤         ) const {
➤      return samples * bytesPerSample * channels;
➤   }
};
```

提交代码并更新Mikado图。

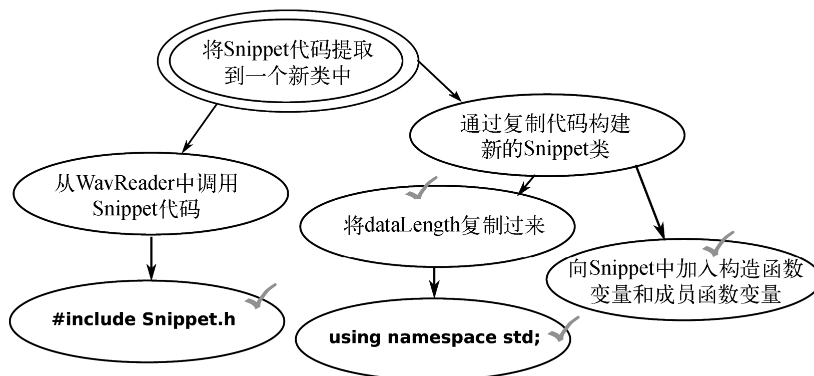


接下来尝试复制writeSnippets()和writeSamples()。我们再一次失败了。错误信息显示未定义成员变量。解决方案是向Snippet.h加入构造函数变量和成员函数变量。可以做出修改并提交吗？当然可以！最终可以将前提条件勾选为已完成。

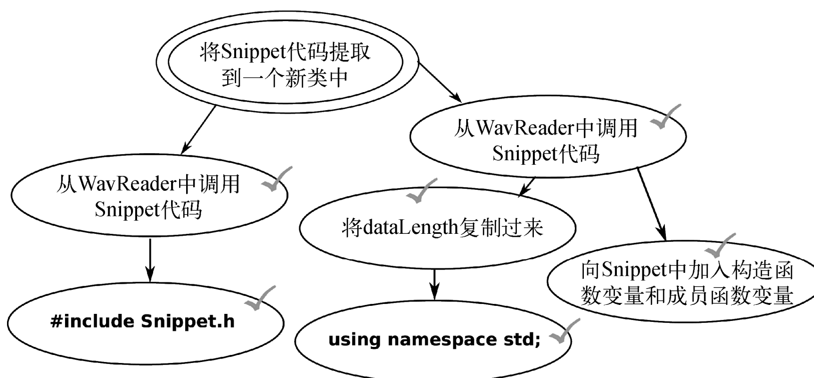
wav/20/Sinppet.h

```
class Snippet {
public:
➤   Snippet(shared_ptr<FileUtil> fileUtil,
➤      shared_ptr<WavDescriptor> descriptor,
➤      const std::string& dest,
➤      rlog::RLogChannel* channel)
➤      : fileUtil_(fileUtil)
➤      , descriptor_(descriptor)
➤      , dest_(dest)
➤      , channel_(channel) { }
// ...
➤ private:
➤   shared_ptr<FileUtil> fileUtil_;
```

```
➤    shared_ptr<WavDescriptor> descriptor_;
➤    const string dest_;
➤    rlog::RLogChannel* channel_;
  };
```



耶！回滚之后，我们能够成功复制writeSnippets()和writeSamples()，然后提交代码。同时，我们可以将最后一个前提目标标记为完成："从WavReader中调用Snippet代码"。（snippet.writeSnippt有点棘手，但我们可以在完成Mikado目标后重新命名这个函数。）



我们的代码还需要一些小的、安全的手工调整。将成员变量channel重命名为channel_。同时为writeSamples()的channels参数提供默认值1，从WavReader.h中复制其定义。即便是很小的改动，其中依然蕴含着一些可以接受的风险（这迫使我们运行一些慢速测试）。

wav/21/Snippet.h

```
class Snippet {
public:
   // ...
   void writeSnippet(
         const string& name, istream& file, ostream& out,
         FormatSubchunk& formatSubchunk,
         DataChunk& dataChunk,
```

```
         char* data
         ) {
      uint32_t secondsDesired{10};
      if (formatSubchunk.bitsPerSample == 0) formatSubchunk.bitsPerSample = 8;
      uint32_t bytesPerSample{formatSubchunk.bitsPerSample / uint32_t{8}};

      uint32_t samplesToWrite{secondsDesired * formatSubchunk.samplesPerSecond};
      uint32_t totalSamples{dataChunk.length / bytesPerSample};

      samplesToWrite = min(samplesToWrite, totalSamples);

      uint32_t totalSeconds { totalSamples / formatSubchunk.samplesPerSecond };

➤     rLog(channel_, "total seconds %i ", totalSeconds);

      dataChunk.length = dataLength(
            samplesToWrite,
            bytesPerSample,
            formatSubchunk.channels);
      out.write(reinterpret_cast<char*>(&dataChunk), sizeof(DataChunk));

      uint32_t startingSample{
         totalSeconds >= 10 ? 10 * formatSubchunk.samplesPerSecond : 0};

      writeSamples(&out, data, startingSample, samplesToWrite, bytesPerSample);

➤     rLog(channel_, "completed writing %s", name.c_str());

      auto fileSize = fileUtil_->size(name);

      descriptor_->add(dest_, name,
            totalSeconds, formatSubchunk.samplesPerSecond, formatSubchunk.channels,
            fileSize);

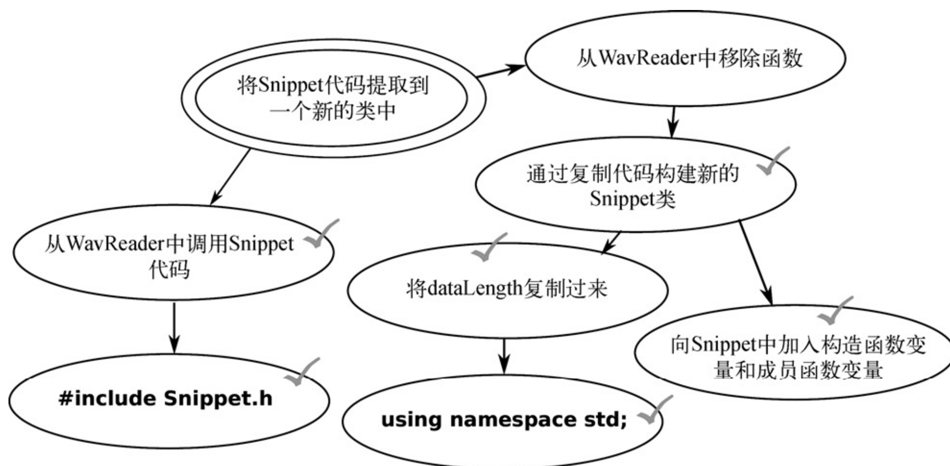
      //out.close(); // ostreams 是 RAII
   }
   void writeSamples(ostream* out, char* data,
         uint32_t startingSample,
         uint32_t samplesToWrite,
         uint32_t bytesPerSample,
➤        uint32_t channels=1) {
➤     rLog(channel_, "writing %i samples", samplesToWrite);
      // ...
   }
};
```

/wav21/WavReader.cpp

```
void WavReader::open(const std::string& name, bool trace) {
   // ...
   auto data = readData(file, dataChunk.length); // 泄漏!

   writeSnippet(name, file, out, formatSubchunk, dataChunk, data);
}
```

任务完成了吗？不完全吧！我们需要删除WavReader中的三个函数，这个步骤还没反映在Mikado图中。我们可以将它作为主目标（“将Snippet代码提取到一个新类中”）的前提目标。

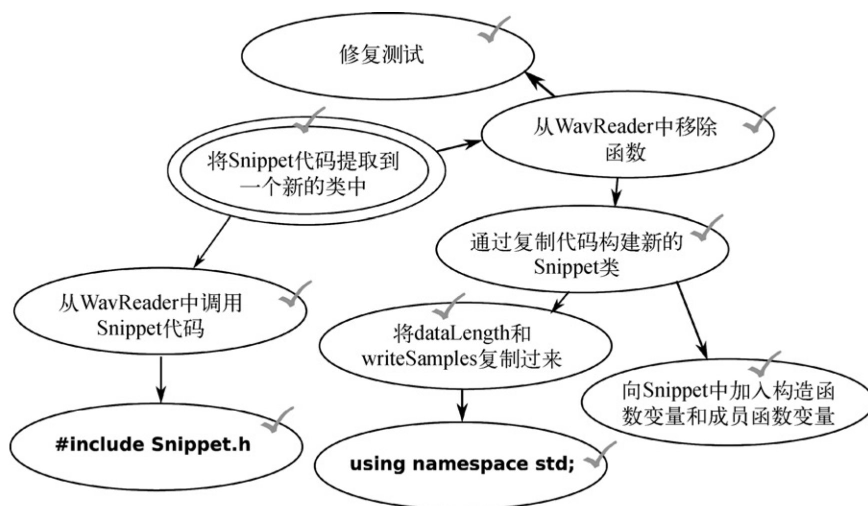


加入新的前提目标并且没有导致失败，这本身并没有错误，但这和写一个自动通过的测试有点类似。早前，我们选择复制代码至Snippet类。或许我们应当着重关注这个问题，并坚持将移动方法作为更直接的目标。

无论如何，我们选择继续前进。由于测试原因，尝试将函数从WavReader中移除失败了。我们加上一个前提目标，修掉测试，勾选这个前提，然后成功地将函数从WavReader中移除，任务完成！（可以为修复测试创建一个前提目标，但现在你无疑已经一览全局了。）

你可以在源码包中找到最终的代码。重要的是要保证最后一个勾选画在了图中的Mikado目标上！（参见下文中的插图。）

从设计的角度来看，我们处于较好的境地。WavReader主要用于读取WAV文件，Snippet主要用于从WAV文件数据写出片段。之前我们还担忧暴露本该私有的函数（如`writeSnippet()`）。现在它们已经是Snippet的公有接口了，我们的担忧也就一扫而空。当然，我们还可以进一步修改，但这一直都会存在。就目前来讲，作为一个增量的改进，这已经足够好了，我们可以继续前进。



带有选中标记的Mikado目标

8.20 有关 Mikado 方法的更多思考

正如前一章示例显示的那样，即便是一个很小的重构，也很容易让人迷失方向、丢三落四。构建Mikado图有助于了解你从哪里来、要往哪里去。

回溯的工作方式是Mikado方法的重要部分。相反，典型的方法会顺着代码，朝着有点模糊的最终目标尝试重构。只有达成目标后，你才能对解决方案有粗略的印象。

前向方法让你可以完成你想要完成的，但这种随意的方法会带来一些问题，很容易出现一无所获的代码改动。此时，你会拒绝重头开始，因为这需要回顾目前为止走过的每一步。大多数时候，你会选择艰难地前行，这可能会将不理想的代码留在原地。

如前文所述，有时你只能到达距最终目标一半路程的地方，并被强制丢弃没有完成的解决方案。不完整的重构工作会让代码看起来令人迷惑。

在一个需要几天或更长时间才能完成的大型重构工作中，Mikado方法能发挥重要作用。因为大部分时间都在探索和分析，所以你可以将真正的工作提炼成简单的脚本，并体现在Mikado图中。此外，不断地尝试各种解决方案以及在遇到问题时及时回滚代码意味着，在方案中重新实施之前的步骤可以提升效率。（在构建Mikado图时，可以加上一些有用的代码段。）

如果拥有一个完整且经过良好实践的Mikado图，那么就能将经过几周探索分析得来的信息提炼为耗时几个小时的脚本。随后，你可以应用这个脚本来实施巨大的、遍及整个系统的改动。你告诉团队成员要确保在下班前做好代码整合并提交，然后应用这个脚本，如果遇到问题，最坏的情况就是回滚代码。你的同事在第二天早上就能获取你提交的大规模改动。这样你的工作对他们

来说影响会很小，反之亦然。

你已经了解了Mikado方法的核心步骤。但是，我强烈建议你进一步学习此技巧。Mikado方法的书籍（*Behead You Legacy Beast: Refactor and Restructure Relentlessly with the Mikado Method* [BE12]）能够提供任何你想知道的知识。同时，它还罗列了应对遗留系统的精彩原则总结。

8.21　这样做值得吗

在经历了本章的遗留代码示例后，你可能会想："天呐！要做这么多事！为什么要这么做呢？"

你已经知道不这么做的代价了。系统自身缓慢、明确的退化无疑会让你痛苦倍增。构建系统的时间变长、分析时间变长、需要更多的时间运行测试，并且维护时间也在变长。

只要能获得支持并专注于系统，使用（包括本章中介绍的）遗留代码管理技巧就可以挽救系统。最初的努力可能只会给代码库带来很小的影响。但如果每个人都秉承做出增量改进的目标，就能开始看到改动带来的巨大回报。（需要多长时间呢？这取决于系统的大小、修改的速度、代码的难度、团队的意愿，等等。）

你的团队必须作出选择：要么弃用遗留系统挽救技巧，并任系统持续退化（这是确定的）；要么放手一搏，决定让事情往好的方向发展。如果选择后者，那么你就要帮助团队列出一些基本规则，以便衡量对每次提交的改动的期望。

8.22　结束语

令人畏惧的遗留代码可以吓跑最好的开发者。你在本章中再一次学到，用增量的方法处理问题非常奏效。只要做少量的安全代码重组，你就能保持前进。不需要测试所有的东西，只要测试需要改动的部分即可。

同时，你也学到了怎样通过Mikado方法做出巨大、遍及整个系统的改动，而且不用承担很大的心理压力。

在下一章中，你将学习如何用TDD来帮助应对软件开发中难度更大的挑战之一，即怎样打造健壮的多线程应用程序。

第 9 章 测试驱动开发与多线程

9.1 开场白

创建一个健壮的多线程应用是一项挑战，需要花费几个小时甚至几天的时间来解决竞争条件和死锁问题。能用测试驱动的方式开发这样的应用程序吗？

答案是肯定的，但是编写处理多线程的测试并不容易。有时，这些测试本身就会产生大量额外的线程，从而在系统中增加了并发复杂性的层面。

9.2 测试驱动开发多线程应用的核心概念

在本章中，你将通过一个示例程序来了解TDD多线程应用的相关核心概念。

分离线程逻辑和程序逻辑。最好的面向对象的设计是尽可能地分离各种关注点。多线程应用程序设计也不例外。多线程是一个关注点，应用程序逻辑是另一个关注点。要尽可能地分离这些关注点，并将耦合降到最低。重申一下，小的方法和类是最好的解决办法（参见6.2.6节）。

*休眠并不好，对吧？*在线程中通过调用`sleep_for()`暂停执行，直到满足相应的条件，是一种糟糕的解决方案。测试的运行会变慢，还会引发随机错误。对失败的定时器测试的反应通常是增加等待时间，这会使测试的平均运行时间变长，更糟糕的是，真正的问题会被隐藏得更深、更久。

简化特定的应用测试至单线程。在引入多线程之前，应用程序代码首先必须能够在单线程环境下工作。一旦引入多线程，仍然要确保应用程序代码能正确地执行。

为特定的应用测试提供消除并发性的方法，可以帮助你保持清醒。从某种意义上来说，测试多线程代码将带你进入集成测试的领域。为了重载线程控制，你可以在应用程序中加入钩子机制，或引入更多的测试。

在引入并发性控制之前，验证并发性问题。在程序中滥用并发控制（Lock和Wait）会极大降低应用程序的性能，甚至有可能解决不了任何真正的并发性问题。接下来的示例程序主要包含了以下核心内容：先编写一个可以演示潜在并发性问题的测试；然后使其恶化，直到测试每次都会

失败。通过演示这些失败，首先能确保继续保持测试驱动的模式。你需要做的仅仅是增加并发性控制，以便测试可以通过。

以测试驱动的方式应对多线程所带来的挑战，仍然需要我们细致地思考和分析线程间是如何交错运行的。遵循前面提到的理念可将麻烦降到最低，从而产生更加清晰的解决方案。

9.3　示例程序 GeoServer

GeoServer为客户端应用程序提供了一些支持，以跟踪大量用户的地理位置（不用担心，这个应用程序不会出售给政府，而且没有你的允许，我是不会泄露你的位置信息的）。典型的客户端应用是基于地图的手机应用程序。我们主要讨论如何搭建服务器端应用，所以你完全可以自己想象客户端应用的模样。

客户端应用注册到服务器端，并开始跟踪用户的位置。客户端会不时地将位置更新信息发送至服务器端。

以下是GeoServer的源代码（包括Location类的头文件）:

c9/1/GeoServerTest.cpp

```
#include "CppUTest/TestHarness.h"
#include "CppUTestExtensions.h"
#include "GeoServer.h"

using namespace std;

TEST_GROUP(AGeoServer) {
   GeoServer server;

   const string aUser{"auser"};
   const double LocationTolerance{0.005};
};
TEST(AGeoServer, TracksAUser) {
   server.track(aUser);

   CHECK_TRUE(server.isTracking(aUser));
}

TEST(AGeoServer, IsNotTrackingAUserNotTracked) {
   CHECK_FALSE(server.isTracking(aUser));
}

TEST(AGeoServer, TracksMultipleUsers) {
   server.track(aUser);
   server.track("anotheruser");

   CHECK_FALSE(server.isTracking("thirduser"));
   CHECK_TRUE(server.isTracking(aUser));
   CHECK_TRUE(server.isTracking("anotheruser"));
```

```
}

TEST(AGeoServer, IsTrackingAnswersFalseWhenUserNoLongerTracked) {
   server.track(aUser);
   server.stopTracking(aUser);

   CHECK_FALSE(server.isTracking(aUser));
}

TEST(AGeoServer, UpdatesLocationOfUser) {
   server.track(aUser);
   server.updateLocation(aUser, Location{38, -104});

   auto location = server.locationOf(aUser);
   DOUBLES_EQUAL(38, location.latitude(), LocationTolerance);
   DOUBLES_EQUAL(-104, location.longitude(), LocationTolerance);
}

TEST(AGeoServer, AnswersUnknownLocationForUserNotTracked) {
   CHECK_TRUE(server.locationOf("anAbUser").isUnknown());
}

TEST(AGeoServer, AnswersUnknownLocationForTrackedUserWithNoLocationUpdate) {
   server.track(aUser);
   CHECK_TRUE(server.locationOf(aUser).isUnknown());
}

TEST(AGeoServer, AnswersUnknownLocationForUserNoLongerTracked) {
   server.track(aUser);
   server.updateLocation(aUser, Location(40, 100));
   server.stopTracking(aUser);
   CHECK_TRUE(server.locationOf(aUser).isUnknown());
}
```

c9/1/GeoServer.h

```
#ifndef GeoServer_h
#define GeoServer_h

#include <string>
#include <unordered_map>

#include "Location.h"

class GeoServer {
public:
   void track(const std::string& user);
   void stopTracking(const std::string& user);
   void updateLocation(const std::string& user, const Location& location);

   bool isTracking(const std::string& user) const;
   Location locationOf(const std::string& user) const;

private:
```

```
   std::unordered_map<std::string, Location> locations_;

   std::unordered_map<std::string, Location>::const_iterator
      find(const std::string& user) const;
};

#endif
```

c9/1/GeoServer.cpp

```
#include "GeoServer.h"
#include "Location.h"
using namespace std;
void GeoServer::track(const string& user) {
   locations_[user] = Location();
}

void GeoServer::stopTracking(const string& user) {
   locations_.erase(user);
}

bool GeoServer::isTracking(const string& user) const {
   return find(user) != locations_.end();
}

void GeoServer::updateLocation(const string& user, const Location& location) {
   locations_[user] = location;
}
Location GeoServer::locationOf(const string& user) const {
   if (!isTracking(user)) return Location{}; // TODO: 性能开销?
   return find(user)->second;
}

std::unordered_map<std::string, Location>::const_iterator
   GeoServer::find(const std::string& user) const {
   return locations_.find(user);
}
```

c9/1/Location.h

```
#ifndef Location_h
#define Location_h

#include <limits>
#include <cmath>
#include <ostream>

const double Pi{ 4.0 * atan(1.0) };
const double ToRadiansConversionFactor{ Pi / 180 };
const double RadiusOfEarthInMeters{ 6372000 };
const double MetersPerDegreeAtEquator{ 111111 };

const double North{ 0 };
const double West{ 90 };
const double South{ 180 };
```

```
const double East{ 270 };
const double CloseMeters{ 3 };

class Location {
public:
   Location();
   Location(double latitude, double longitude);

   inline double toRadians(double degrees) const {
      return degrees * ToRadiansConversionFactor;
   }

   inline double toCoordinate(double radians) const {
      return radians * (180 / Pi);
   }

   inline double latitudeAsRadians() const {
      return toRadians(latitude_);
   }

   inline double longitudeAsRadians() const {
      return toRadians(longitude_);
   }

   double latitude() const;
   double longitude() const;

   bool operator==(const Location& that);
   bool operator!=(const Location& that);

   Location go(double meters, double bearing) const;
   double distanceInMeters(const Location& there) const;
   bool isUnknown() const;
   bool isVeryCloseTo(const Location& there) const;

private:
   double latitude_;
   double longitude_;

   double haversineDistance(Location there) const;
};

std::ostream& operator<<(std::ostream& output, const Location& location);

#endif
```

你可以参考未列在这里的其他源代码文件。嗯……注意文件GeoServer.cpp中的一段注释！有人担心两次访问`locations_`地图会增加性能开销。我们稍后再讨论这个问题（参见10.2节）。

有了这些基本的素材，现在我们来着手充实GeoServer吧。

场景：搜索附近的用户

作为客户端用户，我希望能够标识出地图上某个矩形区域内的所有用户（包括他们的坐标），这样就能知道他们在地图上的位置了。

我们将在GeoServer中实现这一点。

c9/2/GeoServerTest.cpp

```
TEST_GROUP(AGeoServer_UsersInBox) {
   GeoServer server;

   const double TenMeters { 10 };
   const double Width { 2000 + TenMeters };
   const double Height { 4000 + TenMeters};
   const string aUser { "auser" };
   const string bUser { "buser" };
   const string cUser { "cuser" };

   Location aUserLocation { 38, -103 };

   void setup() override {
      server.track(aUser);
      server.track(bUser);
      server.track(cUser);
      server.updateLocation(aUser, aUserLocation);
   }
   vector<string> UserNames(const vector<User>& users) {
      return Collect<User,string>(users, [](User each) { return each.name(); });
   }
};

TEST(AGeoServer_UsersInBox, AnswersUsersInSpecifiedRange) {
   server.updateLocation(
      bUser, Location{aUserLocation.go(Width / 2 - TenMeters, East)});

   auto users = server.usersInBox(aUser, Width, Height);

   CHECK_EQUAL(vector<string> { bUser }, UserNames(users));
}

TEST(AGeoServer_UsersInBox, AnswersOnlyUsersWithinSpecifiedRange) {
   server.updateLocation(
      bUser, Location{aUserLocation.go(Width / 2 + TenMeters, East)});

   server.updateLocation(
      cUser, Location{aUserLocation.go(Width / 2 - TenMeters, East)});

   auto users = server.usersInBox(aUser, Width, Height);

   CHECK_EQUAL(vector<string> { cUser }, UserNames(users));
}
```

c9/2/GeoServer.cp

```
bool GeoServer::isDifferentUserInBounds(
      const pair<string, Location>& each,
      const string& user,
      const Area& box) const {
   if (each.first == user) return false;
   return box.inBounds(each.second);
}

vector<User> GeoServer::usersInBox(
      const string& user, double widthInMeters, double heightInMeters) const {
   auto location = locations_.find(user)->second;
   Area box { location, widthInMeters, heightInMeters };

   vector<User> users;
   for (auto& each: locations_)
      if (isDifferentUserInBounds(each, user, box))
         users.push_back(User{each.first, each.second});
   return users;
}
```

Area表示以某个位置为中心的矩形区域，能够表明该区域是否包含了一个（或其他）位置。以下是Area的头文件。

c9/2/Area.h

```
#ifndef Area_h
#define Area_h

#include "Location.h"

class Area {
public:
   Area(const Location& location, double width, double height);
   Location upperLeft() const;
   Location upperRight() const;
   Location lowerRight() const;
   Location lowerLeft() const;
   bool inBounds(const Location&) const;

private:
   double left_;
   double right_;
   double top_;
   double bottom_;
};

#endif
```

User包含了用户的名字和位置信息。

c9/2/User.h

```
#ifndef User_h
```

```
#define User_h
#include "Location.h"

class User {
public:
   User(const std::string& name, Location location)
      : name_(name), location_(location) {}
   std::string name() { return name_; }
   Location location() { return location_; }

private:
   std::string name_;
   Location location_;
};
#endif
```

9.4 性能要求

我们面临性能方面的考验。产品负责人指出我们期望大量的用户。初始发行版本应该可以同时支持50 000位用户。

我们编写了一个测试，以模拟将用户数目逐步增加至巨大数量级的情况。

c9/3/GeoServerTest.h

```
TEST(AGeoServer_UsersInBox, HandlesLargeNumbersOfUsers) {
   Location anotherLocation{aUserLocation.go(10, West)};
   const unsigned int lots {500000};
   for (unsigned int i{0}; i < lots; i++) {
      string user{"user" + to_string(i)};
      server.track(user);
      server.updateLocation(user, anotherLocation);
   }

   auto users = server.usersInBox(aUser, Width, Height);
   CHECK_EQUAL(lots, users.size());
}
```

当运行测试时，我们注意到，在CppUTest执行HandlesLargeNumbersOfUsers时存在约1.5秒的停顿。虽然看起来并不长，但这仅仅只是一个测试。一个系统中最终可能会有成千上万个测试，即便只有少量运行缓慢的测试，但也会阻碍你按照必要的频率来运行这些测试。我们并不想在快速测试集中运行较慢的HandlesLargeNumbersOfUsers测试。但我们将来仍然有可能需要运行它。完成清理工作之后，最好的措施就是将HandlesLargeNumbersOfUsers移到运行较慢的其他测试集中。

代码看起来运行得很慢，但我们能确定这个测试运行得特别慢吗？现在通过CppUTest的选项-v来重新运行测试。

```
build/utest -v
```

测试运行的输出结果证明新测试是罪魁祸首。

```
TEST(ALocation, ProvidesPrintableRepresentation) - 0 ms
TEST(ALocation, IsNotVeryCloseToAnotherWhenNotSmallDistanceApart) - 0 ms
TEST(ALocation, IsVeryCloseToAnotherWhenSmallDistanceApart) - 0 ms
TEST(ALocation, CanBeAPole) - 0 ms
TEST(ALocation, AnswersNewLocationGivenDistanceAndBearingVerifiedByHaversine) - 0 ms
TEST(ALocation, AnswersNewLocationGivenDistanceAndBearing) - 0 ms
TEST(ALocation, IsNotEqualToAnotherWhenLatAndLongMatch) - 0 ms
TEST(ALocation, IsNotEqualToAnotherWhenLongDiffers) - 0 ms
TEST(ALocation, IsNotEqualToAnotherWhenLatDiffers) - 0 ms
TEST(ALocation, AnswersDistanceFromAnotherInMeters) - 0 ms
TEST(ALocation, IsUnknownWhenLatitudeAndLongitudeNotProvided) - 0 ms
TEST(ALocation, IsNotUnknownWhenLatitudeAndLongitudeProvided) - 0 ms
TEST(ALocation, AnswersLatitudeAndLongitude) - 0 ms
➤ TEST(AGeoServer_UsersInBox, HandlesLargeNumbersOfUsers) - 1689 ms
TEST(AGeoServer_UsersInBox, AnswersOnlyUsersWithinSpecifiedRange) - 0 ms
# ... (results of other tests omitted)

OK (39 tests, 39 ran, 49 checks, 0 ignored, 0 filtered out, 1798 ms)
```

我们仍然不知道真正的问题在于调用`usersInBox()`。或许问题在于测试中用于跟踪和更新50万用户位置信息的Arrange部分。我们需要更精确的测量。

作为临时的调查方案，我们在测试HandlesLargeNumbersOfUsers中声明一个RAII定时器的实例。（要想了解更多关于TestTimer类的信息，参见10.2节。）

c9/4/GeoServerTest.cpp

```
TEST(AGeoServer_UsersInBox, HandlesLargeNumbersOfUsers) {
   Location anotherLocation{aUserLocation.go(10, West)};
   const unsigned int lots {500000};
   for (unsigned int i{0}; i < lots; i++) {
      string user{"user" + to_string(i)};
      server.track(user);
      server.updateLocation(user, anotherLocation);
   }
➤  TestTimer timer;
   auto users = server.usersInBox(aUser, Width, Height);

   CHECK_EQUAL(lots, users.size());
}
```

通过调查方案得知，每次调用`usersInBox()`平均要花费200多毫秒（多次运行测试以确保数据的可靠性）。

```
.............
HandlesLargeNumbersOfUsers elapsed time = 219.971ms
..........................
OK (39 tests, 39 ran, 49 checks, 0 ignored, 0 filtered out, 1823 ms)
```

我们提醒自己，这些信息是相对的，不同机器上的运行结果可能大不相同，但这已足够说明有问题存在。

虽然200毫秒没有一秒或半秒那么严重，但产品负责人对此并不满意。我们和她讨论各种可行方案，一起构造出一个可以满足需求的场景。

场景：搜索附近用户是一个异步的需求

作为客户端用户，我期望“搜索附近用户”的需求能够迅速得到反馈。我想逐个地收到附近用户的信息，每收到一个就在地图上显示出来。

你可能已经意识到这些场景并非是技术上的。现存的指导方针将避免创建不对商业提供可验证价值的场景。“创建数据库表单”或“升级编译器版本”这样的任务可能是最基本的，但你应该只在商业需求的环境执行。有必要从技术角度为上述场景讨论出一个解决方案，且它代表可以传递可论证和可检验的商业价值的特性。

9.5　设计异步方案

通过测试驱动的方式开发程序，你将会得到一个完全不同于初始构想的设计方案。但这并不是完全放弃预先设计的理由。通常情况下，你希望可以基于提供合理方向的路线图开展工作。

不要花太多时间来增加路线图的细节，因为在向前行进时，我们毫无疑问要走一些弯路。路线图中包含过多细节却最终没有执行的部分完全是在浪费时间。

GeoServer客户端需要简单的接口，用于将用户信息、高度、宽度传送给服务器，并从服务器得到用户清单。然而，为了支持异步的用户体验，客户端需要向服务器端传递一个回调函数，以便处理某个区域内接受的所有用户信息。

我们想要分离客户端和多线程的实现细节，以下是我们提议的设计思路。

- 为每一个连接需求创建一个工作项，并将其加入到工作队列中。该工作项包含了用来判断用户是否处在某个区域内的所有信息。
- 在GeoServer端启动一个或多个工作线程。处于空闲状态的工作线程会等待工作队列中可用的工作项。一旦抓取了一个工作项，工作线程就会开始处理该工作项。

可以对GeoServer类中的所有逻辑进行直接编码，但我们不会这么做。如果将多线程和应用程序逻辑杂糅，一旦出现问题，那么将会是一个漫长且令人厌恶的调试过程。而且，几乎所有程序都会出现问题。

因此，我们分离涉及的三个关注点，并用三个类来表征：

- Work类，表示一个工作项；
- ThreadPool类，用于创建工作线程并处理工作队列；
- GeoServer类，用于创建Work对象，并将其发送到ThreadPool以便让其执行。

先从最简单的Work类开始：

c9/5/WorkTest.cpp

```
#include "CppUTest/TestHarness.h"
#include "Work.h"
#include <functional>
#include <vector>
#include <sstream>

using namespace std;

TEST_GROUP(AWorkObject) {
};

TEST(AWorkObject, DefaultsFunctionToNullObject) {
   Work work;
   try {
      work.execute();
   }
   catch(...) {
      FAIL("unable to execute function");
   }
}

TEST(AWorkObject, DefaultsFunctionToNullObjectWhenConstructedWithId) {
   Work work(1);
   try {
      work.execute();
   }
   catch(...) {
      FAIL("unable to execute function");
   }
}

TEST(AWorkObject, CanBeConstructedWithAnId) {
   Work work(1);
   LONGS_EQUAL(1, work.id());
}

TEST(AWorkObject, DefaultsIdTo0) {
   Work work;
   LONGS_EQUAL(0, work.id());
}
TEST(AWorkObject, DefaultsIdTo0WhenFunctionSpecified) {
   Work work{[]{}};
   LONGS_EQUAL(0, work.id());
}

TEST(AWorkObject, CanBeConstructedWithAFunctionAndId) {
   Work work{[]{}, 1};
   LONGS_EQUAL(1, work.id());
}

TEST(AWorkObject, ExecutesFunctionStored) {
   bool wasExecuted{false};
```

```
   auto executeFunction = [&] () { wasExecuted = true; };
   Work work(executeFunction);
   work.execute();
   CHECK_TRUE(wasExecuted);
}

TEST(AWorkObject, CanExecuteOnDataCapturedWithFunction) {
   vector<string> data{"a", "b"};
   string result;
   auto callbackFunction = [&](string s) {
      result.append(s);
   };
   auto executeFunction = [&]() {
      stringstream s;
      s << data[0] << data[1];
      callbackFunction(s.str());
   };
   Work work(executeFunction);
   work.execute();
   CHECK_EQUAL("ab", result);
}
```

c9/5/Work.h

```
#ifndef Work_h
#define Work_h
#include <functional>

class Work {
public:
   static const int DefaultId{0};
   Work(int id=DefaultId)
      : id_{id}
      , executeFunction_{[]{}} {}
   Work(std::function<void()> executeFunction, int id=DefaultId)
      : id_{id}
      , executeFunction_{executeFunction}
      {}
   void execute() {
      executeFunction_();
   }
   int id() const {
      return id_;
   }
private:
   int id_;
   std::function<void()> executeFunction_;
};
#endif
```

基于Work类的测试CanExecuteOnDataCapturedWithFunction仅仅展示了lambda是如何工作的，从技术角度来说，这并不是必需的。测试很快就通过了。这个测试将帮助我们加强理解如何使用lambda，并可以演示客户端代码会如何利用Work类对象。函数`executeFunction`的功能是

抓取本地定义的数据向量，并随后演示这些数据的处理。（截取操作符[&]告诉C++编译器通过引用来抓取任何参考变量。）这个函数也能获取本地定义的回调函数callbackFunction，并将数据向量元素的联接结果发送到该回调函数。

我们最终会删除CanExecuteOnDataCapturedWithFunction测试。当理解了如何用GeoServer代码创建Work类对象时，它将提供一个我们要做的示例程序。

9.6　依然简单的测试驱动

我们开始着手设计ThreadPool类。在引入多线程之前，先通过测试驱动的方式开发一些处理用户需求的模块。

c9/5/ThreadPoolTest.cpp

```
#include "CppUTest/TestHarness.h"
#include "ThreadPool.h"

using namespace std;

TEST_GROUP(AThreadPool) {
   ThreadPool pool;
};

TEST(AThreadPool, HasNoWorkOnCreation) {
   CHECK_FALSE(pool.hasWork());
}

TEST(AThreadPool, HasWorkAfterAdd) {
   pool.add(Work{});
   CHECK_TRUE(pool.hasWork());
}

TEST(AThreadPool, AnswersWorkAddedOnPull) {
   pool.add(Work{1});
   auto work = pool.pullWork();

   LONGS_EQUAL(1, work.id());
}

TEST(AThreadPool, PullsElementsInFIFOOrder) {
   pool.add(Work{1});
   pool.add(Work{2});
   auto work = pool.pullWork();

   LONGS_EQUAL(1, work.id());
}

TEST(AThreadPool, HasNoWorkAfterLastElementRemoved) {
   pool.add(Work{});
   pool.pullWork();
```

```
   CHECK_FALSE(pool.hasWork());
}

TEST(AThreadPool, HasWorkAfterWorkRemovedButWorkRemains) {
   pool.add(Work{});
   pool.add(Work{});
   pool.pullWork();
   CHECK_TRUE(pool.hasWork());
}
```

c9/5/ThreadPool.h

```
#ifndef ThreadPool_h
#define ThreadPool_h

#include <string>
#include <deque>
#include "Work.h"

class ThreadPool {
public:
   bool hasWork() {
      return !workQueue_.empty();
   }

   void add(Work work) {
      workQueue_.push_front(work);
   }

   Work pullWork() {
      auto work = workQueue_.back();
      workQueue_.pop_back();
      return work;
   }

private:
   std::deque<Work> workQueue_;
};
#endif
```

测试AnswersWorkAddedOnPull必须验证抓取的工作项和增加的工作项是相互匹配的。或许可以比较地址，但我们决定为Work对象增加一个身份标志符（ID），从而让测试更简洁。

c9/5/Work.h

```
#ifndef Work_h
#define Work_h
#include <functional>

class Work {
public:
   static const int DefaultId{0};
   Work(int id=DefaultId)
      : id_{id}
```

```
      , executeFunction_{[]{}} {}
   Work(std::function<void()> executeFunction, int id=DefaultId)
      : id_{id}
      , executeFunction_{executeFunction}
      {}
   void execute() {
      executeFunction_();
   }
   int id() const {
      return id_;
   }
private:
   int id_;
   std::function<void()> executeFunction_;
};
#endif
```

9.7 为多线程做好准备

（注意，CppUTest中的内存泄漏检测目前不是线程安全的。你可能想将其关掉，更多信息请参见http://www.cpputest.org/node/25。关掉内存泄漏检测功能会在第8章的编程练习中引入问题。）

我们想让ThreadPool处理抓取并执行工作项。我们需要一个线程。创建的测试展示了如何将Work对象发送到ThreadPool的`add()`函数，并让线程池异步地处理Work对象。

c9/6/ThreadPoolTest.cpp

```
TEST(AThreadPool, PullsWorkInAThread) {
   pool.start();
   condition_variable wasExecuted;
   bool wasWorked{0};
   Work work{[&] {
      unique_lock<mutex> lock(m);
      wasWorked = true;
      wasExecuted.notify_all();
   }};

   pool.add(work);

   unique_lock<mutex> lock(m);
   CHECK_TRUE(wasExecuted.wait_for(lock, chrono::milliseconds(100),
         [&] { return wasWorked; }));
}
```

异步方法意味着函数`add()`可以在工作完成之前，将控制权返还给测试。我们需要找到方法以验证工作项是否被真正执行。我们采取了wait/notify的策略。当创建完一个ThreadPool的实例后，测试程序定义了一个条件变量`wasExecuted`。这个信号量可以避免测试过快完成。

我们创建了一个工作项，该工作项中的回调函数可以设置标志位，并通知所有等待`wasExecuted`条件的线程。我们期望ThreadPool的工作线程来执行这个工作项。测试调用`pool.`

add(work)后，它创建了一个互斥锁，并一直等到相应的标志位被设置。如果没有及时清除条件变量，那么测试将会失败。

在ThreadPool类中增加函数start()用来决定:客户端程序必须表明ThreadPool应该在什么时候启动它的工作线程。因为线程不会在ThreadPool实例化的过程中自动启动，所以，对于之前编写的基于特定应用程序的测试，我们无需担心多线程的问题。函数start()启动了一个工作线程，一旦该线程完成初始化，在析构函数中将会调用join()。

函数worker()等待可用的工作项，然后抓取并执行。

c9/6/ThreadPool.h

```
  #include <string>
  #include <deque>
➤ #include <thread>
➤ #include <memory>

  #include "Work.h"

  class ThreadPool {
  public:
➤    virtual ~ThreadPool() {
➤       if (workThread_)
➤          workThread_->join();
➤    }

➤    void start() {
➤       workThread_ = std::make_shared<std::thread>(&ThreadPool::worker, this);
➤    }
     // ...
  private:
➤    void worker() {
➤       while (!hasWork())
➤          ;
➤       pullWork().execute();
➤    }

     std::deque<Work> workQueue_;
➤    std::shared_ptr<std::thread> workThread_;
  };
```

第一次运行测试后，我们收到了一条错误信息：

```
..terminate called after throwing an instance of 'std::system_error'
  what(): Operation not permitted
```

经过快速的网络搜索及访问Stackoverflow.com网页后，我们在CMakeLists.txt增加了对pthread的链接，从而消除了这个错误。

c9/6/CMakeLists.txt

```
# ...
```

```
    target_link_libraries(utest pthread)
➤ target_link_libraries(utest CppUTest)
```

测试有时能成功地运行。我们只要看一下刻意过度简化的实现，就可以知道问题所在。如果是通过一个命令行脚本反复运行这些测试，你会发现新增加的线程测试时常会触发分段错误。我们将研究另一个方案：用强制手段让错误直接源于测试本身。

9.8　暴露并发性问题

我们想进一步展示：工作线程可以从工作队列中抓取和执行多个工作项。

c9/7/ThreadPoolTest.cpp

```
TEST(AThreadPool, ExecutesAllWork) {
   pool.start();
   unsigned int count{0};
   unsigned int NumberOfWorkItems{3};
   condition_variable wasExecuted;
   Work work{[&] {
      std::unique_lock<std::mutex> lock(m);
      ++count;
      wasExecuted.notify_all();
   }};
   for (unsigned int i{0}; i < NumberOfWorkItems; i++)
      pool.add(work);
   unique_lock<mutex> lock(m);
   CHECK_TRUE(wasExecuted.wait_for(lock, chrono::milliseconds(100),
         [&] { return count == NumberOfWorkItems; }));
}
```

实现中引入了一个`while`循环、一个布尔量标志位，在销毁ThreadPool实例的时候，停止执行`while`循环。

c9/7/ThreadPool.h

```
    #include <string>
    #include <deque>
    #include <thread>
    #include <memory>
➤ #include <atomic>
    #include "Work.h"
    class ThreadPool {
    public:
       virtual ~ThreadPool() {
➤         done_ = true;
          if (workThread_)
             workThread_->join();
       }
       // ...
    private:
       void worker() {
```

```
➤       while (!done_) {
            while (!hasWork())
               ;
            pullWork().execute();
➤       }
     }
➤    std::atomic<bool> done_{false};
     std::deque<Work> workQueue_;
     std::shared_ptr<std::thread> workThread_;
  };
```

遗憾的是……不，幸运的是，蹩脚的实现每次都会导致测试程序瘫痪。在处理多线程时，持续的错误不完全是坏事，它会引导出一个解决方案。分析表明，当测试完成后，ThreadPool的析构函数将标志位done_设为true，并尝试加入线程执行。而线程却无法完成工作，因为它一直陷在while循环中，等待可用的工作项。

在等待工作项的循环中加入一个条件语句，当标志位done_为true的时候，跳出当前的循环。

c9/8/ThreadPool.h

```
  void worker() {
     while (!done_) {
➤       while (!done_ && !hasWork())
➤          ;
➤       if (done_) break;
        pullWork().execute();
     }
  }
```

测试不再导致系统瘫痪，并且在第一次运行时就通过了。然而，这些测试仍然存在间歇性的失败。我们需要进一步找到使测试每次都失败的方法。简单的尝试后发现，增加循环中的工作项数目并不会带来什么差别。测试需要从本身创建的线程中增加工作项。

让我们先重构吧，清除两个往线程池中加入工作项的测试。

c9/9/ThreadPoolTest.cpp

```
TEST_GROUP(AThreadPool_AddRequest) {
   mutex m;
   ThreadPool pool;
   condition_variable wasExecuted;
   unsigned int count{0};
   void setup() override {
      pool.start();
   }

   void incrementCountAndNotify() {
      std::unique_lock<std::mutex> lock(m);
      ++count;
      wasExecuted.notify_all();
   }
```

```
   void waitForCountAndFailOnTimeout(
         unsigned int expectedCount,
         const milliseconds& time=milliseconds(100)) {
      unique_lock<mutex> lock(m);
      CHECK_TRUE(wasExecuted.wait_for(lock, time,
            [&] { return expectedCount == count; }));
   }
};

TEST(AThreadPool_AddRequest, PullsWorkInAThread) {
   Work work{[&] { incrementCountAndNotify(); }};
   unsigned int NumberOfWorkItems{1};

   pool.add(work);
   waitForCountAndFailOnTimeout(NumberOfWorkItems);
}

TEST(AThreadPool_AddRequest, ExecutesAllWork) {
   Work work{[&] { incrementCountAndNotify(); }};
   unsigned int NumberOfWorkItems{3};

   for (unsigned int i{0}; i < NumberOfWorkItems; i++)
      pool.add(work);

   waitForCountAndFailOnTimeout(NumberOfWorkItems);
}
```

在ThreadPool类中，从析构函数中抽取部分代码放在单独的函数中。

c9/9/ThreadPool.h

```
  virtual ~ThreadPool() {
➤    stop();
  }

➤ void stop() {
➤    done_ = true;
➤    if (workThread_)
➤       workThread_->join();
➤ }
```

9.9　在测试中创建客户端线程

这次我们假设错误是由围绕工作队列争夺数据造成的。主线程将工作项加入到工作队列中，函数`pullWork()`将工作项从队列中移除，且工作线程不断查询工作队列中是否有可用的工作项。

我们的测试不是简单的失败，而是会导致分段错误。工作队列的并行更新看起来是一个疑点。为了修补这个问题，我们先编写一个可以稳定产生同样错误的测试。

c9/10/ThreadPoolTest.cpp

```
TEST_GROUP(AThreadPool_AddRequest) {
```

```
      mutex m;
      ThreadPool pool;
      condition_variable wasExecuted;
      unsigned int count{0};

➤     vector<shared_ptr<thread>> threads;

      void setup() override {
         pool.start();
      }

➤     void teardown() override {
➤        for (auto& t: threads) t->join();
➤     }
➤     // ...
   };
   // ...
   TEST(AThreadPool_AddRequest, HoldsUpUnderClientStress) {
      Work work{[&] { incrementCountAndNotify(); }};
      unsigned int NumberOfWorkItems{10};
      unsigned int NumberOfThreads{10};

      for (unsigned int i{0}; i < NumberOfThreads; i++)
         threads.push_back(
             make_shared<thread>([&] {
                for (unsigned int j{0}; j < NumberOfWorkItems; j++)
                  pool.add(work);
             }));
      waitForCountAndFailOnTimeout(NumberOfThreads * NumberOfWorkItems);
   }
```

测试通过`for`循环来创建任意数目的工作线程。测试将线程句柄存储在向量中，这样就可以正确地等待所有线程结束。可以在测试集的`teardown()`函数中找到这些代码。

将`NumberOfThreads`和`NumberOfWorkItems`设为1，然后出现了同样的间歇性错误。试验了一些组合后我们发现，如果10个线程中的每个线程发送10个请求，那么会导致分段错误。

我们在函数`hasWork()`、`add()`和`pullWork()`中加入了锁保护。这些保护利用互斥对象`lockObject_`来防止多个线程同时访问被保护的代码。

c9/11/ThreadPool.h

```
   #include <string>
   #include <deque>
   #include <thread>
   #include <memory>
   #include <atomic>
➤ #include <mutex>

   #include "Work.h"

   class ThreadPool {
   public:
```

```
   // ...
   bool hasWork() {
➤     std::lock_guard<std::mutex> block(mutex_);
      return !workQueue_.empty();
   }

   void add(Work work) {
➤     std::lock_guard<std::mutex> block(mutex_);
      workQueue_.push_front(work);
   }

   Work pullWork() {
➤     std::lock_guard<std::mutex> block(mutex_);

      auto work = workQueue_.back();
      workQueue_.pop_back();
      return work;
   }
   // ...
   std::atomic<bool> done_{false};
   std::deque<Work> workQueue_;
   std::shared_ptr<std::thread> workThread_;
➤  std::mutex mutex_;
};
```

测试通过了。将线程的数目增加到200，情况看起来依然很好。

9.10　在 ThreadPool 中创建多个线程

就这样结束了吗？已经解决了所有的并发性漏洞吗？为了找出遗留的问题，一种分析方法是想想任何存在的缺口。缺口是指，我们基于某个“事实”做了一些假设，但因为其他失控线程的干扰，这个“事实”并不成立。

`worker()`函数看起来是唯一可能包含有潜在风险代码的函数。在`worker()`函数中，我们轮询可用的工作项，每次都是通过循环来建立、释放`hasWork()`中的一个锁。一旦有可用的工作项，循环退出，控制权移交给`pullWork().execute()`。设想一下，如果在这个极为短暂的时间段里，其他的线程抓取了可用工作项，那么会发生什么情况？

目前，ThreadPool仅仅管理一个线程，这意味着`worker()`函数通过串行的方式，一个接一个地抓取并执行可选工作项，没有机会引发并发性的问题。我们在ThreadPool中加入对线程池的支持，让其名副其实。

c9/12/ThreadPoolTest.cpp

```
TEST(AThreadPoolWithMultipleThreads, DispatchesWorkToMultipleThreads) {
   unsigned int numberOfThreads{2};
   pool.start(numberOfThreads);
   Work work{[&] {
      addThreadIfUnique(this_thread::get_id());
```

```
      incrementCountAndNotify();
   }};
   unsigned int NumberOfWorkItems{500};

   for (unsigned int i{0}; i < NumberOfWorkItems; i++)
      pool.add(work);

   waitForCountAndFailOnTimeout(NumberOfWorkItems);
   LONGS_EQUAL(numberOfThreads, numberOfThreadsProcessed());
}
```

测试DispatchesWorkToMultipleThreads展示了客户端代码可以启动指定数量的线程。为了验证ThreadPool是否真正在不同线程中处理任务，先要更新回调函数：如果ID是唯一的，那么就增加一个线程。断言比较指定的线程数目和被执行的线程数目。

（遗憾的是，这个测试在某些极端情况下会失败，因为其中一个线程会处理所有的工作项。如何去除潜在的影响因素就留给读者思考了。）

修改ThreadPool以支持产生指定数量的线程，仅仅需要管理线程对象的向量。

c9/12/ThreadPool.h

```
  #include <string>
  #include <deque>
  #include <thread>
  #include <memory>
  #include <atomic>
  #include <mutex>
➤ #include <vector>

  #include "Work.h"

  class ThreadPool {
  public:
     // ...
     void stop() {
        done_ = true;
➤       for (auto& thread: threads_) thread.join();
     }

➤    void start(unsigned int numberOfThreads=1) {
➤       for (unsigned int i{0u}; i < numberOfThreads; i++)
➤          threads_.push_back(std::thread(&ThreadPool::worker, this));
➤    }
     // ...
  private:
     // ...
     std::atomic<bool> done_{false};
     std::deque<Work> workQueue_;
     std::shared_ptr<std::thread> workThread_;
     std::mutex mutex_;
➤    std::vector<std::thread> threads_;
  };
```

测试总是失败。考虑到对worker()函数的怀疑，我们增加一行代码来处理并不存在的工作项（换句话说，其他线程已经抓取了该工作项）。

c9/12/ThreadPool.h

```
   Work pullWork() {
      std::lock_guard<std::mutex> block(mutex_);

➤     if (workQueue_.empty()) return Work{};

      auto work = workQueue_.back();
      workQueue_.pop_back();
      return work;
   }
```

测试通过了，事情变得明朗了。我们的修复直接解决了并发性问题。

对并发性问题的保守反应是，在出现问题的地方增加同步机制。最终可能会导致测试运行变慢。相反，测试驱动能帮助我们确定仅在需要的地方加入同步机制。

9.11 回到 GeoServer

我们已经设计并实现了ThreadPool，现在开始使用这个类。第一步是修改usersInBox()，增加监听或者回调函数的参数。更新代码，将User对象通过回调函数返回给客户端，这样就可以异步地集中这些对象。

在测试中，监听程序仅仅记录那些被传进updated()回调函数的用户。

c9/13/GeoServerTest.cpp

```
TEST(AGeoServer_UsersInBox, AnswersOnlyUsersWithinSpecifiedRange) {
➤  class GeoServerUserTrackingListener: public GeoServerListener {
➤  public:
➤     void updated(const User& user) { Users.push_back(user); }
➤     vector<User> Users;
➤  } trackingListener;

   server.updateLocation(
      bUser, Location{aUserLocation.go(Width / 2 - TenMeters, East)});

➤  server.usersInBox(aUser, Width, Height, &trackingListener);

➤  CHECK_EQUAL(vector<string> { bUser }, UserNames(trackingListener.Users));
}
```

c9/13/GeoServer.h

```
class GeoServerListener {
public:
   virtual void updated(const User& user)=0;
};
```

```
class GeoServer {
public:
   // ...
   std::vector<User> usersInBox(
         const std::string& user, double widthInMeters, double heightInMeters,
         GeoServerListener* listener=nullptr) const;
   // ...
};
```

c9/13/GeoServer.cpp

```
vector<User> GeoServer::usersInBox(
      const string& user, double widthInMeters, double heightInMeters,
      GeoServerListener* listener) const {
   auto location = locations_.find(user)->second;
   Area box { location, widthInMeters, heightInMeters };

   vector<User> users;
   for (auto& each: locations_)
      if (isDifferentUserInBounds(each, user, box)) {
         users.push_back(User{each.first, each.second});
➤        if (listener)
➤           listener->updated(User{each.first, each.second});
      }
   return users;
}
```

和往常一样，我们寻求逐步改进的方式，留下直接返回用户向量的逻辑。这可以在浪费大量时间为其他测试编写类似实现之前，证明我们的想法。

我们更新AnswersOnlyUsersWithinSpecifiedRange，以及被忽略的、慢速的测试HandlesLargeNumbersOfUsers。将GeoServerUserTrackingListener类的通用声明作为要素加入测试集。我们删除了支持直接返回用户向量的代码。最后，我们修改`usersInBox()`以假定GeoServerListener是有效的指针。请参考code/c9/14中的clean-up部分。

GeoServer的测试AnswersUsersInSpecifiedRange和AnswersOnlyUsersWithinSpecifiedRange应该仍然可以工作。但如果使用ThreadPool，我们需要在测试中引入等待机制，就像在ThreadPoolTest编写的测试一样。相反，我们选择引入测试替身，将ThreadPool的`add()`函数的实现简化为单线程。

c9/15/GeoServerTest.cpp

```
TEST_GROUP(AGeoServer_UsersInBox) {
   GeoServer server;
   // ...
   class SingleThreadedPool: public ThreadPool {
   public:
      virtual void add(Work work) override { work.execute(); }
   };
   shared_ptr<ThreadPool> pool;
```

```
   void setup() override {
      pool = make_shared<SingleThreadedPool>();
      server.useThreadPool(pool);
      // ...
   }
   // ...
};
TEST(AGeoServer_UsersInBox, AnswersUsersInSpecifiedRange) {
   pool->start(0);
   server.updateLocation(
      bUser, Location{aUserLocation.go(Width / 2 - TenMeters, East)});
   server.usersInBox(aUser, Width, Height, &trackingListener);
   CHECK_EQUAL(vector<string> { bUser }, UserNames(trackingListener.Users));
}
```

我们将ThreadPool的`add()`函数定义为虚函数，以允许重载。

测试显式地启动线程池，这是设计上的考虑——客户端程序负责启动线程池（这个重要的协议将在你编写的单独测试中得到最好的体现）。

c9/15/GeoServer.h

```
class GeoServer {
public:
   // ...
   void useThreadPool(std::shared_ptr<ThreadPool> pool);
   // ...
};
```

c9/15/GeoServer.cpp

```
  void GeoServer::usersInBox(
        const string& user, double widthInMeters, double heightInMeters,
        GeoServerListener* listener) const {
     auto location = locations_.find(user)->second;
     Area box { location, widthInMeters, heightInMeters };
     for (auto& each: locations_) {
        Work work{[&] {
           if (isDifferentUserInBounds(each, user, box))
              listener->updated(User{each.first, each.second});
        }};
➤       pool_->add(work);
     }
  }

➤ void GeoServer::useThreadPool(std::shared_ptr<ThreadPool> pool) {
➤    pool_ = pool;
➤ }
```

需要编写一个和多线程池交互的测试吗？为了测试驱动或普通单元测试，不需要！我们已经演示了，ThreadPool可以接收工作项并将其分派给不同的线程；GeoServer逻辑可以判断用户能否在矩形区域内正确地工作；也演示了GeoServer将工作项发送到ThreadPool的逻辑。

任何更进一步的测试都应该是其他类型的，仅在需要时才编写。因为之前使用多线程的兴趣在于判断我们是否能从`usersInBox()`得到及时反馈，以及异步的返回位置信息，所以我们确实需要一个测试。

我们增加一个类似于HandlesLargeNumbersOfUsers的新测试，但在单独的线程中触发`usersInBox()`，并在主线程中等待所有的回调。我们想将这个测试放在慢测试集中。

c9/17/GeoServerTest.cpp

```
TEST_GROUP_BASE(AGeoServer_ScaleTests, GeoServerUsersInBoxTests) {
   class GeoServerCountingListener: public GeoServerListener {
   public:
      void updated(const User& user) override {
         unique_lock<std::mutex> lock(mutex_);
         Count++;
         wasExecuted_.notify_all();
      }

      void waitForCountAndFailOnTimeout(unsigned int expectedCount,
            const milliseconds& time=milliseconds(10000)) {
         unique_lock<mutex> lock(mutex_);
         CHECK_TRUE(wasExecuted_.wait_for(lock, time, [&]
                  { return expectedCount == Count; }));
      }
      condition_variable wasExecuted_;
      unsigned int Count{0};
      mutex mutex_;
   };
   GeoServerCountingListener countingListener;
   shared_ptr<thread> t;

   void setup() override {
      pool = make_shared<ThreadPool>();
      GeoServerUsersInBoxTests::setup();
   }

   void teardown() override {
      t->join();
   }
};

TEST(AGeoServer_ScaleTests, HandlesLargeNumbersOfUsers) {
   pool->start(4);
   const unsigned int lots{5000};
   addUsersAt(lots, Location{aUserLocation.go(TenMeters, West)});

   t = make_shared<thread>(
         [&] { server.usersInBox(aUser, Width, Height, &countingListener); });

   countingListener.waitForCountAndFailOnTimeout(lots);
}
```

9

（考虑到之前为`usersInBox()`编写的测试中有很多类似的设置过程，你在这里看到的代码是一个经过大量重构后的代表。对于wait/notify概念的实现和使用，GeoServerCountingListener和ThreadPoolTest中仍然有大量重复的代码。我们将重构一个任何线程测试都能使用的构造体。）

9.12　结束语

编写多线程代码或许是软件开发领域中最复杂的任务。支持并发性使其变得更难，好在很少需要经常编写这样的代码。TDD模式可以让你的多线程编程沿着科学方法论的方向前行，远离神秘主义。不同于希望你深入理解并发执行的机理，或者在任何出现问题的地方放置锁和同步机制，我们希望你可以将TDD模式作为一个工具，用来验证或否定有关并发性问题的假设。

我们已经覆盖了所有关于TDD的主要知识点。你已经学会了如何用测试的方式驱动开发；如何进行必要的测试；如何解决具体的多线程问题。下一步你将学习有关TDD模式的其他知识点。

第 10 章

测试驱动开发的其他概念和讨论

10

10.1 开场白

并不是所有事情都能放进小巧、整洁的框架中。在前面的章节中，你学习了TDD的核心理念：TDD的周期、TDD的基本概念、构建的指导方针、创建和使用测试替身的方式、设计要素、编写质量测试的方式以及如何处理遗留代码的挑战。你还学习了如何通过测试驱动的方式编写多线程代码。从编写代码的角度来说，差不多就是这些内容……但还有其他一些零碎的事情。

你将在本章中学习以下并未在其他章节展开讨论的内容。

- **TDD和性能**：消除对性能的担忧可以让你在练习测试驱动期间睡个好觉。
- **集成测试和验收测试**：还需要什么类型的测试？它们和单元测试的区别是什么？
- **变换优先级假设**（Transformation Priority Premise，TPP）：这是决定你如何编写下一个测试的正式方法。
- **三角法**：这是驱动编写通用代码的一个技巧（虽然其本身就可以作为一个主题，但我们仍将其包含在TPP章节中）。
- **首先编写断言部分**：编写测试的推荐方法。

10.2 测试驱动开发与性能

在任何系统中，合格的性能是一项非常重要的需求。考虑到性能方面的潜能，很多人可能倾向于使用C++编程。本书的很多地方有意规避了性能方面的担忧，并让你们参考这一节。这并不意味着性能不重要，相反，它很重要。

归入到性能测试集中的绝大多数内容既不是TDD，也不是单元测试。本节展示了以测试为中心的策略可以优化性能；讨论了单元级测试如何帮助实现这个策略，以及设计与性能的关联；并强调在尝试解决性能问题之前，应该要先找寻最优设计方案。

一般来说，性能方面的考虑不是功能上的需求。在最多有10 000个用户同时在线的负载下，系统需要在半秒内对用户交互作出反应。系统需要整夜运行一批处理脚本，等等。这些都是集成层面的关注点（详见10.3节），需要一个集成且部署好的系统。你无法用验证独立逻辑的单元测试来测试这些关注点。

从（单元级）测试驱动的角度来看，你几乎不可能基于已有的知识说出，“这个函数必须在5微秒或更短的时间内作出反应”这样的话。在决定需求之前，你需要了解函数的性能特征是如何与终端行为的需求联系起来的。即使能得到一个具体而微观的性能需求，你将会发现：由于不同机器间的不同特性，你很难找到一个一致的、支持所有平台（开发、集成、产品，等等）的测试手段。

10.2.1　性能优化测试的策略

以下是性能优化的通用策略。

- 依据测试框架构建并运行驱动代码，由此得到系统性能的基准值。
- 确保测试能正确展示特性的功能——优化系统时很容易破坏系统的功能性。
- 将驱动代码转变为可以指定当前性能基准的测试。如果某个尝试的优化导致性能下降，那么这个*基准测试*将会失败。
- 新增一个*目标测试*以运行相同的功能，只有性能达到一定要求时测试才会通过。（这可能是基准测试中的第二个断言。）
- 确定性能瓶颈。
- 尝试优化与性能瓶颈相关的代码。你应该有能力判断是否存在可能的算法优化（比如，将复杂度为$O(n^2)$的算法替换为$O(n \log n)$）。如果是这样的话，那么就从这里开始吧。否则，就从高质量设计和良好表达性的角度入手。通常而言，未能以最佳方式利用C++可能是潜在的因素（比如，如何传递参数、使用参数、构造新的对象，被误导尝试去做的比STL容器或Boost更好）。
- 确保单元测试和验收测试依然可以通过。
- 运行基准测试；如果测试失败（换句话说，如果新的性能更糟），那么就放弃当前的改动，然后继续尝试。
- 运行目标测试，如果测试通过，那就发布吧！
- 如果目标测试失败，可以尝试通过以下方式解决性能问题：找出第二大的性能瓶颈，尝试提升性能，以此类推。然而，你的优化尝试可能并不合适。更好的方法是记录相对的性能提升，并为代码的改动做上标记。继续找寻另一个优化方案，重复这一过程，检查是否达到了性能目标。

　如果你是通过增量合并的方式实现优化，那么要确保及时更新基准测试的条件。

以下是尝试做优化时的一些非常重要的概念。

- 在与生产目标具有同样特性的机器上运行性能测试。在其他机器上的测试结果可能不能准确地反映出优化对产品的影响，因此，进行这些优化可能会浪费时间，甚至导致性能变得更糟糕。
- 不做任何假设。对需要优化的地方的直觉往往是错误的。永远要比对优化前后的测试数据。
- 首先要正确地设计，且仅在此基础上采取优化措施。除非万不得已，否则不要为了引入某些优化而牺牲系统的可维护性和可读性。记住，首先要正确地设计！

10.2.2 相关单元级性能测试

单元级性能测试能帮助你沿着TDD的方向前行，但无法利用它们来判断是否已经达到性能目标要求。相反，把它们视为工具将帮助你探究这些难题。

你将在本节中学习获取函数平均运行时间的一个简单技巧。只有这些优化都针对同一个函数时，执行时间才有意义。

在一些罕见的情况下，你可以预先定义单元级的需求，然后利用相关的单元级性能测试（Relative Unit-Level Performance Tests，我称它们为RUPT）来测试驱动这个需求。否则，你将置身于开发之后再测试的领域。

以下是关于RUPT的一些步骤。

(1) 创建一个循环，多次（如50 000次）反复执行需要对其进行时间测量的特性。应该消除由启动开销或时钟周期带来的任何差异。还要确认编译器没有优化这些需要计时的特性。

(2) 在循环前获取当前的时间戳，并将其存储在变量`start`中。

(3) 在执行行为的代码后获取当前的时间戳，并将其存储在变量`stop`中。相对测量值就是`stop`和`start`之间的时间差。

(4) 运行RUPT并记录时间差。调整循环次数以寻找几秒的时间差。

(5) 以数量级的方式增加循环次数。运行测试并确保时间差也是以类似的方式增长。如果不是这样，那么就说明RUPT不能准确地表征你的优化尝试。你需要找到原因并修复。

(6) 继续多次运行RUPT。如果时间差的变化非常大，说明这不是一个有效的RUPT。找到原因并修复，否则，记录平均时间差。

(7) 尝试优化代码。

(8) 多次运行RUPT并记录平均时间差。

(9) 如果能获得较大的性能改善，则继续运行性能测试，并调整基准目标。否则，抛弃这些改动。

将相关的单元级性能测试作为一种探测手段，你可以决定抛弃一段犹如烂泥般毫无意义的代码，或者进一步利用它。在任何情况下，这些代码都不应该出现在你的产品级单元测试集中。

10.2.3 尝试优化 GeoServer 代码

现在我们来学习一个创建RUPT的简短例子：

c9/24/GeoServerTest.cpp

```
TEST(AGeoServer_Performance, LocationOf) {
   const unsigned int lots{50000};
   addUsersAt(lots, Location{aUserLocation.go(TenMeters, West)});

   TestTimer t;
   for (unsigned int i{0}; i < lots; i++)
      server.locationOf(userName(i));
}
```

TestTimer是一个简单的类，一旦超出范围，它将在控制台打印出性能测量结果。关于TestTimer的具体实现，请参见10.2.4节。

以下是我们测试的代码。`locationOf()`和`isTracking()`都调用了find，这是一个不可接受的性能下降吗？

c9/24/GeoServer.cpp

```
bool GeoServer::isTracking(const string& user) const {
   return find(user) != locations_.end();
}

Location GeoServer::locationOf(const string& user) const {
   if (!isTracking(user)) return Location{}; // TODO性能开销？

   return find(user)->second;
}
```

将循环次数设为50 000并多次运行测试。我们记录下平均运行时间（在我的机器上约为50毫秒）。

将循环次数扩大为500 000并多次运行测试，然后记录平均运行时间。我们希望看到平均运行时间大致成比例地增长，事实也确实如此。平均运行时间是574毫秒。如果没有增长，我们需要找到一种方式阻止C++编译器来优化循环中的某些操作。（在gcc中可以增加一条汇编指令：`asm("");`）。

用修改代码消除对`find()`函数的第二个调用。

c9/25/GeoServer.cpp

```
Location GeoServer::locationOf(const string& user) const {
   // 优化后
   auto it = find(user);
   if (it == locations_.end()) return Location{};
   return it->second;
}
```

没错，添加注释是明智之举（虽然你可能要提供更多解释）。熟悉TDD的优秀程序员总会不断地尝试提高代码的质量。如果没有注释来描述你为什么如此编写代码，那么一位优秀的程序员很可能会清除掉这段关于性能优化的代码。而且，因为不会频繁地运行和性能相关的验收测试，所以当你发现目标性能测试失败时，可能很难甄别出这是由哪些代码改动导致的。

重新运行性能测试并记录下新的平均运行时间——488毫秒，这比之前快了86毫秒。数字显示，冗余地调用`find()`函数会导致约18%的性能下降。这听起来好像很多，但记住，我们运行了500 000个需求，对于每一个需求，差别是0.17微秒。

这些事实从性能角度上来说是关于行为中的变化。虽然它们仅仅提供了相对的、孤立的价值，但它们并不是猜测。我们知道，代码优化的尝试是成功的，因为它提高了代码片段的执行效率。这比我们之前了解得更多，也使很多开发者在尝试优化一个方案之后了解得更多。

现在的问题是，它是有用的吗？基于这一点，我们将运行基准性能测试和目标性能测试，并判断这个优化是否是必需的。如果该优化不是必需的，只是让代码变得更复杂，那么就删除这个优化。

保留该优化的代价看起来很小。`locationOf()`函数仅仅将一行代码变成了三行代码。很多有用的优化往往会增加很多相当难以解释和维护的代码。

因为有清晰的设计，所以另一个潜在的优化应该很容易实现。在同时跟踪成千上万位用户的GeoServer中，增加用户缓存是十分合理的方案。在任何给定的时间段内，服务器有可能会被询问一小部分用户的位置信息，并且很多需求将会和之前的需求重复。目前都是通过访问子函数`find()`来查找`location_`地图。我们可以修改`find()`函数中的代码来使用缓存。客户端代码依然保留当前的设计。相反，在一个总是直接访问成员变量的类中引入缓存机制将是一个持久的过程。

清晰的设计从两个方面对性能优化有所帮助。首先，当拥有简洁的函数定义时，用分析工具更容易查明性能问题所在。其次，简洁的类和函数可以帮助你思考出富有创意的优化方法。一旦找出问题所在，改动代码也会变得更轻松。相反，想像一下，在一个有500行代码的函数中隐藏了一个性能瓶颈。你将花费更多时间找出问题并修复。（而且，一个有500行代码的函数几乎不可能拥有充分的测试来让你有信心做一些适当的性能优化。）

10.2.4　TestTimer类

TestTimer类是一个草草写就的简单工具，可在测试中任何需要的地方使用。在函数结束运行之前，它将打印出函数运行时间，以及传递给构造函数的解释性文本。如果使用无参数的构造函数，那么解释性文本就是当前测试的名称。

c9/25/TestTimer.h

```
#ifndef TestTimer_h
```

```
#define TestTimer_h

#include <string>
#include <chrono>

struct TestTimer {
   TestTimer();
   TestTimer(const std::string& text);
   virtual ~TestTimer();

   std::chrono::time_point<std::chrono::system_clock> Start;
   std::chrono::time_point<std::chrono::system_clock> Stop;
   std::chrono::microseconds Elapsed;
   std::string Text;
};

#endif
```

c9/25/TestTimer.cpp

```
#include "TestTimer.h"
#include "CppUTest/Utest.h"
#include <iostream>

using namespace std;

TestTimer::TestTimer()
   : TestTimer(UtestShell::getCurrent()->getName().asCharString()) {
}

TestTimer::TestTimer(const string& text)
   : Start{chrono::system_clock::now()}
   , Text{text} {}
TestTimer::~TestTimer() {
   Stop = chrono::system_clock::now();
   Elapsed = chrono::duration_cast<chrono::microseconds>(Stop - Start);
   cout << endl <<
      Text << " elapsed time = " << Elapsed.count() * 0.001 << "ms" << endl;
}
```

你可以并且应该增强Timer类以满足需求。你可能想将它变成线程安全的（它并不是）。你可能偏向于使用一些不同的、但与平台相关的时间度量API，或者你的系统可以提供一个C++11的高分辨率时钟的单独实现。你或许有能力使用更小的时间度量单位（纳秒！）或者更大的时间度量单位。又或者你可以选择直接将这几行代码插入到测试中，虽然这看起来是无谓的举动。

10.2.5　性能和小函数

C++程序员常常担心调用成员函数的性能开销。因此，很多程序员不愿意接受创建小的函数和类的观点。“我不想将代码抽取出来放在另一个单独的函数中，这样做可能会引入性能问题。”但现今的编译器是非常聪明的，其在很多方面的优化比手工的更好。

与其基于那些无稽之谈而抵制使用小函数，还不如看看具体的数据。

c9/26/GeoServer.cpp

```
Location GeoServer::locationOf(const string& user) const {
   // 优化后
   auto it = locations_.find(user);
   if (it == locations_.end()) return Location{};
   return it->second;
}
```

在将find()变成内联函数之前，平均运行时间是488毫秒。内联之后的平均运行时间是476毫秒。从统计的角度来看，500 000次运行导致的差异几乎微不足道。

是不是find()函数从一开始就被编译器变成内联函数了呢？如果强制让gcc不去内联该函数，那么在运行时间上并无实质的区别（474毫秒）。

c9/27/GeoServer.h

```
std::unordered_map<std::string, Location>::const_iterator
   find(const std::string& user)
➤  const
   __attribute__((noinline));
```

小函数的另一个有趣方面是，C++编译器很有可能在一开始就将其变成内联函数。对较大的函数来说，这实际上减少了编译器优化代码的机会。

事实上，不将部分代码抽取出来并放进较小的函数中代表着不好的设计，而且它实际上并不能提高应用程序的性能。性能方面的专家早就知道这一点。

而且，不要盲目地相信我，我可能是个婆婆妈妈的人。要相信你自己的测量结果。

10.2.6　推荐

性能优化方面的很多想法都是基于民间传说和其他人的经验。不要相信其他人的经验。其次，既然几乎每个人都在说同一件事情，那么听听大家的一致说法是很有价值的。我会将我的一些经验加入其中。

> **我的优化经验**
>
> 作为程序员，我参与了许多与优化相关的工作。作为顾问，我和很多程序员一起合作过，而优化就是他们的主要任务（其中一个平台需要每秒处理20 000多项交易）。在这两个领域中，我经历并见证过源于严格方法的成功案例，这些方法和之前的推荐不谋而合。我也目睹过一个令人扼腕的失败案例：一个公司花费高薪聘请了一些咨询专家，竭力尝试应对一个实时大规模应用所带来的挑战，却采用了“东试一下，西试一下”的方法。
>
> 以下是应对性能挑战的一些关键因素。

- 一个稳固的架构：**架构**一词意味着，一旦就绪就不要轻易改变方方面面的布局。例如，各个模块之间的流程关系图（分布在服务器端和客户端），架构如何在不修改代码的基础上支持不同配置的硬件（换句话说，通过扩充硬件的方式）？
- 一个拥有简洁代码、稳固且灵活的设计，以及在需要大规模修改代码时能提供灵活性的测试。
- 一组从一开始就定义好的、关于未来性能期望的性能目标测试。如果你希望在前期将应用程序发布给十几位用户，而最终用户数量达到一百位，你需要立即知道在新的代码中是否因用户数目的变化而存在风险。

就代码级优化的趋势而言，我看到一些迹象，听到一些性能专家反驳“先设计，然后仅在需要时才优化”的观点。

我看到过许多执迷不悟的优化尝试。有些情况是基于被误导或明显错误的民间传说（有时甚至是另一种语言）。在另一些情况中，优化建议曾经是正确的，但后来被改进后的编译器和运行时间所淘汰。

有些代码级的优化的确可以归到**自由**的范畴。比如，在C++中，用引用传递比用值传递的效率更高，且不影响代码的表达性。如果这些优化没有降低可读性或易维护性，那就去看看吧。否则，就留待以后（很久以后）再研究吧。

10.3　单元测试、集成测试和验收测试

TDD模式是帮助程序员增量设计和开发代码的一种实践。你已经学会了如何使用它，并通过编写单元测试来验证C++中的一些逻辑，这也允许你不断修正代码设计。

在本书中，单元指的是一个小而独立，且能影响某些系统行为的逻辑片段。定义中的**独立**表明，你可以单独运行这个逻辑单元。这要求去除该逻辑单元对其他模块（如服务调用、应用程序接口、数据库、文件系统，等等）的依赖关系。（从技术的角度来看，独立的代码应该和其他代码没有任何耦合；但实际的方法表明，并不总是需要所有的代码逻辑都是完全独立的。）就像本书其他地方重点说明的一样（如4.3节），单元测试对于驱动开发测试的重要意义在于，它能快速修改和调整。

不难发现，仅仅靠单元测试还是不够的。因为它们仅仅验证了那些细小的、相互独立的代码片段，无法验证端对端或已部署方案的正确性。除了单元测试，系统还需要其他测试，这些测试能让你对即将发布的产品的质量抱有很高的信心，其中包含了系统测试、客户测试、验收测试、负载测试、性能测试、可用性测试、功能测试以及可扩展性测试（有些测试大同小异）。因为所有这些测试都是基于集成的软件产品进行验证的，所以都是集成测试。

负责定义集成测试的人往往取决于不同的情况。典型情况下无外乎三种人：测试员、程序员和客户。

根据敏捷开发社区，客户测试是指那些用以验证软件是否满足商业需求的测试。在一个敏捷过程中，测试是在程序开发之前就定义好的，这样就可以给研发团队提供各种各样的规范，这和TDD模式很类似。敏捷开发的支持者往往将客户测试预先定义为*验收测试*。如果研发团队开发的软件能够通过所有的测试，那么客户就会同意接受这款软件。

10.3.1　测试驱动开发如何与验收测试建立联系

通过预先定义验收测试的方式来驱动系统的开发流程，以及通过使用TDD来驱动一些逻辑模块的开发流程，这二者是非常类似的。实际上，采用验收测试的团队称这种流程为*验收测试驱动开发*（Acceptance Test–Driven Development，ATDD）。

你可以看看验收测试是如何融入开发流程的、使用了哪些工具、这些测试应该是什么样的，等等。如果你理解了TDD，那么你已经具备了绝大部分的知识来理解什么是ATDD，以及如何在实际中成功地应用它。

TDD与ATDD的重要区别在于，谁定义了这些测试以及谁会使用这些测试。对TDD而言，程序员负责用编程语言定义单元测试。因此，即将阅读或者使用这些测试的人毫无疑问就是程序员本身（但这并不意味着你可以毫不关心他们代码的可阅读性）。

对ATDD而言，客户（可能也包含了产品负责人或商业分析员）负责根据商业需求定义验收测试。他们绝不是凭空捏造。创建健壮的验收测试，需要团队包括测试员和程序员在内的其他人员的意见与信息。每个人都会使用基于ATDD的测试。

有一些专门论述ATDD的书籍，如《实例化需求：团队如何交付正确的软件》、《验收测试驱动开发：ATDD实例详解》。通过搜索关键字，你也可以找到一些相关的其他信息。

10.3.2　程序员定义的集成测试

作为程序员，你总是可以选择编写面向程序员的集成测试。一些精挑细选的集成测试是非常有价值的，而且公司可能也需要。可以直接测试数据访问层以获得更多的即时信息以及关于代码和数据储存定义之间差异性的故障信息，也可以使用一组冒烟测试来快速判断部署的配置是否正确。

然而，集成测试很难维护。因为集成测试应对的软件被部署在定制的环境中，需要和外部服务和数据存储的野蛮世界进行交互，所以它们是脆弱的。始终保持测试最新且可以在所有的部署环境中运行是巨大的挑战。

最好只编写所需要的集成测试，而不是越多越好。作为一个策略，要么尝试将集成测试定位为客户测试（你的客户愿意采纳），要么移除它的依赖关系，并将其作为展示代码级逻辑的单元测试。

如果你的团队已经通过单元测试工具（如Google Test)创建了一些测试，你可能发现很多表面上的单元测试实际上是集成测试。这些测试尝试验证部分代码逻辑，但由于和其他逻辑模块之间的耦合，它们被纳入慢而脆弱的集成测试中。

抽出时间分类集成测试。对每个测试采取以下四个动作中的一个：

- 清理集成测试，并将其作为验收测试出售；
- 通过移除对其他逻辑模块的依赖，将其转变为一个快速的单元测试；
- 将其作为勉强保留的集成测试；
- 删除它。

立即删除快速单元测试集中的任何遗留集成测试。

10.3.3　测试驱动开发和验收测试驱动开发的重合部分

当团队同时基于TDD和ATDD开展工作时，一个令人焦虑的事情是有些测试不可避免地会出现重叠现象，尤其是当所有任务都是正确的、通过测试驱动模式完成的情况下。绝大多数验收测试通常代表一些功能上的兴趣点，并且展示了系统如何和其他系统进行交互。在做TDD时，你也将通过测试驱动用户接口层的开发。这是重复的劳动吗？

的确，基于接口层的测试看起来非常类似。然而，测试的受众和目的是完全不同的。没有人会阅读你的程序员测试。相反，设计验收测试就是让任何人都可以读取。

客户定义的测试将针对部分端对端功能提供大范围的覆盖率。不能期望这些测试会覆盖所有可能的排列组合，因为这会使测试运行时间变得非常长。可以通过TDD模式编写单元测试，竭力快速通过一组数量庞大的排列组合。

从设计的角度来说，重复的代码量应该要很小。在设计良好的系统中，用户接口层非常薄，主要作为商业领域类（这里应该还有许多非用户接口层的类）的代表。因此，针对这些用户接口层的单元测试，仅仅需要验证任务被正确地授权。

单纯地从测试的角度来看，同时拥有单元测试和验收测试的好处是，提供了一层额外的保护。你的单元测试不可避免地无法覆盖重要的场景。在单元级之上创建一组由不同人员参与定义的测试，将会带来有价值的安全保障。编写一些测试来表征商业模式中没考虑到的场景，反之亦然（但不要忘了，网是由很多孔洞组成的）。

缺陷意味着有错误的或遗漏掉的单元测试。可以通过测试驱动的方式弥补缺陷。先编写一个可以重现故障的测试，然后修改代码使单元测试（以及任何相关的其他验收测试）通过。

10.4　变换优先级假设

本书中曾提过，你编写的下一个测试，就是对系统有微量增强的那一个。如果严格遵循TDD的流程，尝试在继续前进之前演示测试故障，那么你将体会到这对大步迈进的意义（详见3.5节）。遵循简单设计的最后一条规则——最小化类和方法的数量（参见6.2节）——将有助于避免过度设计，而仅编写需要的代码。TDD的第三条规则（详见3.4节）也提到过，只需要编写能使测试通过的代码。

小步骤非常重要，因为在系统逐步增强的过程中，较大的步骤将浪费更多时间。构造一个过分渲染、完全超出测试需求的方案需要从大段代码中详细检查相关测试。

通过增量演进来发展系统是成功的TDD需要具备的条件。经常备份以及需要时尝试不同的方式，将加快进度。（你需要一个好的版本管理工具来支持修改。）

Robert C. Martin的TPP是决定下一个测试的另一个工具，TPP提出了变换优先级的一个列表。变换表示代码从特定性到增加些许一般性。利用TPP，你可以进行测试的最高优先级变换。前提是要遵循TPP可以增量地演进、发展系统。TPP的使用有助于避免测试驱动中的一些糟粕。

你可以在网址http://web.archive.org/web/20130113152824/和http://cleancoder.posterous.com/the-transformation-priority-premise中找到最初的优先级列表。优先级列表并不是非常简单的技巧，而是前提条件。其他的博客提出了略做变形的优先级列表。

10.4.1　了解变换

TPP听起来有些复杂。具体的例子胜过千言万语。我们将从TDD的角度逐步深入Soundex：用初始排列顺序的优先级变换列表（Transform Priority List, TPL）编写的第一个例子。

变　换	说　明
({}→nil)	将没有实现的代码替换为nil
(nil→constant)	将nil替换为一个常量
(constant→constant+)	将一个简单常量替换为一个复杂常量
(constant→scalar)	将一个常量替换为一个变量或参数
(statement→statements)	增加无条件语句
(unconditional→if)	分离执行分支
(scalar→array)	将一个变量/参数替换为一个数组
(array→container)	将一个数组替换为一个复杂容器
(statement→recursion)	将一个语句替换为递归调用
(if→while)	将一个条件判断语句替换为循环
(expression→function)	将一个表达式替换为一个函数
(variable→assignment)	替换一个变量的值

（资料来源：http://en.wikipedia.org/wiki/Transformation_Priority_Premise）

鉴于我们已经解析过Soundex，接下来将重点讨论变换相关的代码，其他部分会一笔带过。第一个测试略有不同，我们将处理填充字符"0"：

tpp/1/SoundexTest.cpp

```
TEST(SoundexEncoding, AppendsZerosToWordForOneLetterWord) {
   Soundex soundex;
   auto encoded = soundex.encode("A");

   CHECK_EQUAL("A000", encoded);
}
```

编译测试的失败代表着我们对变换的第一个需求——从没有任何代码到返回nil，这是位于TPL顶端的最简单变换。实现返回nil值的encode()才能使编译test通过……现在从失败的单元测试开始：

tpp/1/Soundex.h

```
class Soundex {
public:
   std::string encode(const std::string& word) const {
      return nullptr;
   }
};
```

通过将nil变换为常数，我们修补了单元测试错误，这是TPL中的第二项。

tpp/2/Soundex.h

```
class Soundex {
public:
   std::string encode(const std::string& word) const {
➤     return "A000";
   }
};
```

10.4.2　三角法

之前创建Soundex时，我们去除了硬编码的常量"A"，将其作为重构的一个步骤。为了消除产品代码中的重复字符串，引入一个变量。同时，我们决定删除产品代码中的特定硬编码值，让它和测试名RetainsSoleLetterOfOneLetterWord中蕴含的目标一致。我们在操作时遵循了TPP精神：通过增量方式逐步地一般化代码。TPL中的每个变换代表着从特定性到增加些许一般性的变化。

这次，我们将使用*三角法*去除代码中的硬编码。《测试驱动开发》中首次提出了三角法。通过增加第二个测试用例，三角法从不同的角度接近了相同的行为。

tpp/3/SoundexTest.cpp

```
TEST(SoundexEncoding, AppendsZerosToWordForOneLetterWord) {
   CHECK_EQUAL("A000", soundex.encode("A"));
```

```
➤   CHECK_EQUAL("B000", soundex.encode("B"));
}
```

引入`if`语句可以让失败的测试通过：如果单词的首字母是A，则返回`"A100"`，否则就返回`"B100"`。引入的代码表示变换(unconditional→if)。我们选择优先级更高的变换(constant→scalar)。

tpp/3/Soundex.h

```
class Soundex {
public:
   std::string encode(const std::string& word) const {
      return word + "000";
   }
};
```

10.4.3　浏览测试列表

下一个测试在哪？我们想尽可能用最高优先级的变换来增加代码。让我们看看列表里还剩下哪些测试：

```
PadsWithZerosToEnsureThreeDigits
ReplacesConsonantsWithAppropriateDigits
ReplacesMultipleConsonantsWithDigits
LimitsLengthToFourCharacters
IgnoresVowelLikeLetters
IgnoresNonAlphabetics
CombinesDuplicateEncodings
UppercasesFirstLetter
IgnoresCaseWhenEncodingConsonants
CombinesDuplicateCodesWhen2ndLetterDuplicates1st
DoesNotCombineDuplicateEncodingsSeparatedByVowels
```

现在有一个无条件语句用到了一个常量和一个标量。因为无需担心从{}、nil、array和if的变换，所以就减少了和TPL的关联部分。

如果需要处理子字符串、字符串长度，或将字母变成大写，任何代码都需要引入一次函数调用。(或许，如果带着创意去思考，不需要函数调用的方式可能也行得通。)暂时先绕开看上去需要函数调用的测试，查找其他更高优先级的测试。

大多数不需要函数调用的测试可能需要条件语句。在引入条件判断代码之前，硬编码特定常量只能到这一步了。我们先处理编码：

tpp/4/SoundexTest.cpp

```
TEST(SoundexEncoding, ReplacesConsonantsWithAppropriateDigits) {
   CHECK_EQUAL("A100", soundex.encode("Ab"));
}
```

用（unconditional→if）变换让代码变得更加通用。

tpp/4/Soundex.h

```
class Soundex {
public:
   std::string encode(const std::string& word) const {
➤     std::string code("");
➤     code += word[0];
➤     if (word[1])
➤        code += "100";
➤     else
➤        code += "000";
➤     return code;
   }
};
```

使用TPP时可以稍微放松要求，避免过早地禁锢代码。你仍然要移除重复的代码，保持良好的表达性，但可以暂时先留下简单的if语句和while循环。你会发现，避免使用复杂形式（如三元运算符和for循环）更利于代码重构。

我们的方案过于生搬硬套，但这并不是什么问题。它并不清晰，而且有重复的代码，这是个问题，我们需要重构。

tpp/5/Soundex.h

```
class Soundex {
public:
   std::string encode(const std::string& word) const {
      std::string code("");
➤     code += head(word) + encodeTail(word);
➤     return zeroPad(code);
   }

➤  char head(const std::string& word) const {
➤     return word[0];
➤  }

➤  std::string encodeTail(const std::string& word) const {
➤     if (word[1] == 0) return "";
➤     return "1";
➤  }

➤  std::string zeroPad(const std::string& code) const {
➤     if (code[1] != 0)
➤        return code + "00";
➤     return code + "000";
➤  }
};
```

zeroPad()中的代码还是有些糟糕，不是吗？下面进行第二轮重构：

tpp/6/Soundex.h

```
std::string zeroPad(const std::string& code) const {
➤  return code + (hasEncodedCharacters(code) ? "00" : "000");
```

```
   }

➤ bool hasEncodedCharacters(const std::string& code) const {
➤    return code[1] != 0;
➤ }
```

我刚才提到了哪些像三元运算符这样的的奇妙构想？看来删除一些重复代码是合理的。如果它带来任何问题，我们可以在备份之后将其删除。

增加一个断言以便对另一个辅音字母编码。在实现方面，我们可以引入一个`if`语句，但这只会引入一个复制的构造体，不会使代码变得更加通用。寻找下一个最高优先级的变换：（标量→数组）。

tpp/7/SoundexTest.cpp

```
TEST(SoundexEncoding, ReplacesConsonantsWithAppropriateDigits) {
   CHECK_EQUAL("A100", soundex.encode("Ab"));
   CHECK_EQUAL("A200", soundex.encode("Ac"));
}
```

tpp/7/Soundex.h

```
class Soundex {
public:
➤    Soundex() {
➤       codes_['b'] = "1";
➤       codes_['c'] = "2";
➤    }
     // ...
     std::string encodeTail(const std::string& word) const {
        if (word[1] == 0) return "";
➤      return codes_[static_cast<size_t>(word[1])];
     }
     // ...
➤ private:
➤    std::string codes_[128];
};
```

上述代码完成了辅音字母列表，还对表达性做了一点重构。

tpp/8/Soundex.h

```
class Soundex {
public:
   Soundex() {
➤     initializeCodeMap();
   }

   void initializeCodeMap() {
      codes_['b'] = codes_['f'] = codes_['p'] = codes_ ['v'] = "1";
      codes_['c'] = codes_['g'] = codes_['j'] = codes_ ['k'] =
         codes_['q'] = codes_['s'] = codes_['x'] = codes_['z'] = "2";
      codes_['d'] = codes_['t'] = "3";
      codes_['l'] = "4";
```

```
        codes_['m'] = codes_['n'] = "5";
        codes_['r'] = "6";
     }
     // ...

     std::string encodeTail(const std::string& word) const {
        if (word[1] == 0) return "";
➤       return codeFor(word[1]);
     }

➤    std::string codeFor(char c) const {
➤       return codes_[static_cast<size_t>(c)];
➤    }

     // ...
  };
```

再一次浏览列表中剩下的测试。大部分看起来仍然需要引入一次函数调用。比函数调用优先级更高的是支持循环的两个变换：一个通过`while`循环，另一个通过递归。测试ReplacesMultiple-ConsonantsWithDigits看起来需要一个循环方案。

TPL的之后版本包含了一些优先级变更。我们用的是最初的版本，在这个版本中，递归比循环优先级更高。其重点已经变成了辩论的话题。在功能性语言（如Erlang或者Clojure）中，你可能需要一个递归的解决方案。在C++中，选择哪种方案完全取决于你自己。你可能会比较递归方案和循环方案的性能。

我们将继续使用初始版本中的TPL排序，看看接下来会发生什么。

通过TPP的方式逐步增加代码，使每一步都是微小、增量的变化。我们并没有要求大规模地修改现有代码，对递归的引入同样如此。

tpp/9/SoundexTest.cpp

```
TEST(SoundexEncoding, ReplacesMultipleConsonantsWithDigits) {
   CHECK_EQUAL("A234", soundex.encode("Acdl"));
}
```

tpp/9/Soundex.h

```
  std::string encode(const std::string& word) const {
     std::string code("");
➤    code += head(word);
➤    encodeTail(word, code);
     return zeroPad(code);
  }
  // ...
➤ void encodeTail(const std::string& word, std::string& code) const {
     if (word[1] == 0) return;
➤    code += codeFor(word[1]);
➤    encodeTail(tail(word), code);
  }
➤ std::string tail(const std::string& word) const {
```

```
➤    return word.substr(1);
➤ }
```

还存在两个问题。首先，它不能正常工作。我们需要调整进行编码的字符'0'的数量。其次，测试调用了函数substr()，我们选择引入递归，因为它比函数调用的优先级更高。

TPP是前提和正在开展的工作，但不是解决所有编程挑战的灵丹妙药，也不是一组严格的规定。TPL的目的是帮助你找到下一个最小增量。在例子中，递归方案表明它是一个良好的增量步骤，这一点非常重要。对优先级规则来说，引入函数调用的方案看起来是可以接受的（递归方案处理集合—字符串仅仅是一组字符的集合—常常需要从其尾部抽取数据的函数，通常这就是最佳方案）。

解决字符填充问题的方法有很多种，其中之一就是初始化一个用来表征需要填充字符'0'的计数器或字符串，然后在函数encodeTail()每次添加编码字符时，递减计数器或字符串。然而，这需要一条赋值语句，而赋值语句在优先级列表中处于低级。TPL中更简单、优先级更高的方法是，在函数zeroPad()中考虑字符串的长度。

tpp/10/Soundex.h

```
const static size_t MaxCodeLength{4};
std::string zeroPad(const std::string& code) const {
➤    return code + std::string(MaxCodeLength - code.length(), '0');
}
```

我们意识到函数encodeTail()可以“反过来”用，如果调用时传入字符串的尾部（而不是完整的字符串），那么它可以操作字符串的第一个字符。修改允许我们做一些额外的重构以增强代码的表达性。

tpp/11/Soundex.h

```
std::string encode(const std::string& word) const {
   std::string code(1, head(word));
   encode(tail(word), code);
   return zeroPad(code);
}
void encode(const std::string& word, std::string& code) const {
   if (word.empty()) return;
   code += codeFor(head(word));
   encode(tail(word), code);
}
const static size_t MaxCodeLength{4};
std::string zeroPad(const std::string& code) const {
➤    return code + std::string(MaxCodeLength - code.length(), '0');
}
```


我们太喜欢这些可以确定改动能够工作的测试了！

核心算法是严谨且清晰的。正如上次编写Soundex一样，函数encode()清晰地表明了对字符串编码的策略。我们来看看其他的测试。这里选择了IgnoresVowelLikeLetters，它看起来只需要引

入if语句。

tpp/12/SoundexTest.cpp

```
TEST(SoundexEncoding, IgnoresVowelLikeLetters) {
   CHECK_EQUAL("B234", soundex.encode("BAaEeIiOoUuHhYycdl"));
}
```

未作任何修改的情况下，测试通过了！像往常一样，我们需要思考一下原因（参见3.5节），但现在这不是重点。如果要想严格遵守TPP，前提是它指导我们将最小的增量逐步合并到代码中。过早通过的测试表明，编写大量代码的可能性不大。

测试通过的原因是codes_数组对并不包含的元素总是返回null。添加null对代码没有任何改变。接着我们发现，IgnoresNonAlphabetics也是基于同样原因而每次都通过。

现在引入CombinesDuplicateEncodings。

tpp/13/SoundexTest.cpp

```
TEST(SoundexEncoding, CombinesDuplicateEncodings) {
   CHECK_EQUAL(soundex.codeFor('f'), soundex.codeFor('b'));
   CHECK_EQUAL(soundex.codeFor('g'), soundex.codeFor('c'));
   CHECK_EQUAL(soundex.codeFor('t'), soundex.codeFor('d'));
   CHECK_EQUAL("A123", soundex.encode("Abfcgdt"));
}
```

测试运行时发生了错误——std::length_error。快速查看回溯（在Linux下使用gdb指令），发现问题出现在函数zeroPad() 中。如果一个字符串包含三个以上的字符，那么zeroPad()会尝试创建一个由'0'组成的字符串，但字符串的长度是负数。

这是否意味着应该将焦点切换到LimitsLengthToFourCharacters？TPP给出了否定的答案，因为这需要调用函数length()，而CombinesDuplicateEncodings仅仅需要一个条件语句。然而，程序抛出的异常终止了测试的执行，无法看到测试的直接故障。我们决定先找到失败的测试，记录下来，碰到问题时再尝试调整代码。禁用这个测试，然后开始运行LimitsLengthToFourCharacters。

tpp/14/SoundexTest.cpp

```
TEST(SoundexEncoding, LimitsLengthToFourCharacters) {
   CHECK_EQUAL(4u, soundex.encode("Dcdlb").length());
}
```

一个小小的改动（expression→function）让测试通过了。

tpp/14/Soundex.h

```
  void encode(const std::string& word, std::string& code) const {
➤    if (word.empty() || isFull(code)) return;
     code += codeFor(head(word));
     encode(tail(word), code);
  }
➤ bool isFull(std::string& code) const {
```

```
➤     return code.length() == MaxCodeLength;
➤ }
```

再次引入CombinesDuplicateEncodings，不出所料，测试失败了。测试将字符串的头传递给递归函数`encode()`，将其作为基准，然后与当前添加的字符进行比较。为了避免和函数`head()`的名称产生冲突，我们为测试选取了另一个名字。

tpp/15/Soundex.h

```
std::string encode(const std::string& word) const {
   std::string code(1, head(word));
➤  encode(tail(word), code, head(word));
   return zeroPad(code);
}
void encode(const std::string& word, std::string& code,
➤     char H) const {
   if (word.empty() || isFull(code)) return;
➤  std::string digit = codeFor(head(word));
➤  if (digit != codeFor(H))
➤     code += codeFor(head(word));
➤  encode(tail(word), code, head(word));
}
```

我们的方案需要重构。因为要用字符串的头部和尾部在函数`encode()`中编码，所以我们传递了整个字符串。同时，选择一个不会引起冲突的名称，添加一个辅助函数，以说明测试的用途。（因为`encode()`函数的每行代码都被修改了，所以并没有重点突出，新增加的`isSameEncodingAsLast()`也是如此。）

tpp/16/Soundex.h

```
std::string encode(const std::string& word) const {
   std::string code(1, head(word));
➤  encode(word, code);
   return zeroPad(code);
}
void encode(const std::string& word, std::string& code) const {
   auto tailToEncode = tail(word);
   if (tailToEncode.empty() || isFull(code)) return;

   auto digit = codeFor(head(tailToEncode));
   if (isSameEncodingAsLast(digit, word))
      code += digit;
   encode(tailToEncode, code);
}
bool isSameEncodingAsLast(
      const std::string& digit,
      const std::string& word) const {
   return digit != codeFor(head(word));
}
```

类似的测试CombinesDuplicateCodesWhen2ndLetterDuplicates1st也应该需要大致相同的变换。（我们将首字母小写的字符串指定为断言的期望值，因为还没有处理在Soundex编码时首字母大写

的情况。）

tpp/17/SoundexTest.cpp

```
TEST(SoundexEncoding, CombinesDuplicateCodesWhen2ndLetterDuplicates1st) {
   CHECK_EQUAL("b230", soundex.encode("bbcd"));
}
```

测试通过了，因为已经让递归函数`encode()`处理了整个编码过程。同样，另一个类似的测试也通过了。

tpp/18/SoundexTest.cpp

```
TEST(SoundexEncoding, DoesNotCombineDuplicateEncodingsSeparatedByVowels) {
   CHECK_EQUAL("J110", soundex.encode("Jbob"));
}
```

所有的测试都通过了，我们反倒有些不安，因此，通过另一个场景减轻我们的担心，它也通过了。

tpp/18/SoundexTest.cpp

```
TEST(SoundexEncoding, CombinesMultipleDuplicateEncodings) {
   CHECK_EQUAL("J100", soundex.encode("Jbbb"));
}
```

（还有一种可能的场景：基于H和W可能会被区别对待这样的事实，这取决于你和谁交流。因为上次构造Soundex时忽略了这个潜在的区别，所以我们将继续忽略，让事情变得简单。）

先大写首字母，这也需要更新CombinesDuplicateCodesWhen2ndLetterDuplicates1st的期望值。

tpp/19/SoundexTest.cpp

```
TEST(SoundexEncoding, CombinesDuplicateCodesWhen2ndLetterDuplicates1st) {
   CHECK_EQUAL("B230", soundex.encode("bbcd"));
}
TEST(SoundexEncoding, UppercasesFirstLetter) {
   CHECK_EQUAL("A", soundex.encode("abcd").substr(0, 1));
}
```

这个变换虽然简单，但使用了函数调用。（可以直接编写`::toupper()`的代码，但我并不认为TPP允许做这件事情。）你已经看过`upper()`的实现，这里就不列举了。

tpp/19/Soundex.h

```
std::string encode(const std::string& word) const {
➤  std::string code(1, toupper(head(word)));
   encode(word, code);
   return zeroPad(code);
}
```

嗯。最后一个测试触发了一点思考，如果输入字符串的字母相同，但第一个字母大写，而第二个字母小写，那么会怎么样呢？

tpp/20/SoundexTest.cpp

```
TEST(SoundexEncoding, IgnoresCaseWhenEncodingConsonants) {
   CHECK_EQUAL(soundex.encode("BCDL"), soundex.encode("bcdl"));
}
```

啊哈！出错了，以下是一个快速解决方案：

tpp/20/Soundex.h

```
std::string codeFor(char c) const {
   return codes_[static_cast<size_t>(lower(c))];
}
```

完成了吗？还没添加PadsWithZerosToEnsureThreeDigits。它直接通过了，但选择它的目的是为了文档记录。

tpp/21/SoundexTest.cpp

```
TEST(ASoundexEncoding, PadsWithZerosToEnsureThreeDigits) {
   CHECK_EQUAL("I000", soundex.encode("I"));
}
```

第二次总是显得容易些，但这次的成功更多地归功于TPP流程的使用，而非对问题的理解。我们做了一些主观选择，可能没有严格遵循TPP，但最终结果说明了一切。

以下是TPP测试驱动的核心算法：

tpp/21/Soundex.h

```
std::string encode(const std::string& word) const {
   std::string code(1, toupper(head(word)));
   encode(word, code);
   return zeroPad(code);
}

void encode(const std::string& word, std::string& code) const {
   auto tailToEncode = tail(word);
   if (tailToEncode.empty() || isFull(code)) return;

   auto digit = codeFor(head(tailToEncode));
   if (isSameEncodingAsLast(digit, word))
      code += digit;

   encode(tailToEncode, code);
}
```

以下是非TPP测试驱动的核心算法：

c2/40/Soundex.h

```
std::string encode(const std::string& word) const {
   return stringutil::zeroPad(
      stringutil::upperFront(stringutil::head(word)) +
         stringutil::tail(encodedDigits(word)),
```

```
        MaxCodeLength);
}

std::string encodedDigits(const std::string& word) const {
   std::string encoding;
   encodeHead(encoding, word);
   encodeTail(encoding, word);
   return encoding;
}

void encodeHead(std::string& encoding, const std::string& word) const {
   encoding += encodedDigit(word.front());
}

void encodeTail(std::string& encoding, const std::string& word) const {
   for (auto i = 1u; i < word.length(); i++)
      if (!isComplete(encoding))
         encodeLetter(encoding, word[i], word[i - 1]);
}
void encodeLetter(std::string& encoding, char letter, char lastLetter) const {
   auto digit = encodedDigit(letter);
   if (digit != NotADigit &&
         (digit != lastDigit(encoding) || charutil::isVowel(lastLetter)))
      encoding += digit;
}
```

我知道自己应该维护哪个版本。TPP不仅能够产生更简洁的算法，花费的精力也更少。

尽管很好，但TPP仅仅是一个前提。我越用它，就越对产出感到开心。它是一个高级的课题，是深入理解TDD的美味大餐。

TPP要求预先思考TDD流程中的每个路径。必须考虑以下几点。

- 当前测试的哪个实现具有更高优先级？能想出更富创意的方法吗？
- 相对于即将着手的下一个测试，是否存在比其优先级更高的测试？
- 其余的测试是什么？对TPP而言，维护一个测试列表（参见2.3节）显得尤为重要。

正如本书中介绍的一样，即使不引入TPP，TDD也能顺畅地工作。但是使用TPP会做得更加出色。

10.5　编写断言

和其他的TDD实践者交流的越多，就越会发现有很多实现TDD的方法。假设你幸运地避免了关于“唯一正确方法”的激烈争吵，例如，你会发现有些开发者极力推崇“每个测试对应一个断言”，而其他人认为这是一个过分的目标。你会发现有些实践者坚持认为，测试的名称应该遵循同样的格式，而其他开发者根本不关心这个。

这里有对错之分吗？绝大多数时候你会发现，站在对立立场的人们对每一个论点都有深刻而

理性的思考。

你应该已经在本书中注意到了我的风格，毫无疑问，你会觉得一些事情比另一些事情更引人入胜。我的建议是，在轻易丢弃之前，尝试一些性质不同或你不认同的事情。你可能会发现惊喜。很早之前我就回避了“一个测试对应一个断言”（参见7.3节），坚持这个原则的话，在99%的时候你会发现其价值所在。

基本上，除了遵循TDD流程和产出高质量代码，其余的事情更主要在于风格和偏好。请记住，寻找更好的做事方式是你的职责。不管对我的风格，还是你自己风格的感受如何，希望不要在两年之后发现彼此仍在用同样的方式工作。

10.5.1　断言—行为—排列

TDD中“红—绿—重构”的循环顺序是固定的，但在测试中编写这些语句的顺序却是灵活的。许多开发者采取由上而下的开发方式。他们首先编写排列部分，其次是行为部分，最后是断言。这种方法没有什么错误（恰巧我也常常采用这种方法），但如果先编写断言部分，或许会更好。

到目前为止，你已经习惯了针对不断完善的产品代码编写测试程序。为不存在的测试编写断言的想法似乎并不应该让人感到过于震惊。但为什么要这么做呢？

先编写断言部分使你思考增加这些行为的目的，并进一步推动你来描述这对已经完成的目标意味着什么。如果这一步很困难，很可能是因为你还没有掌握足够的信息来继续编写测试。

更重要的是，先编写断言部分有助于从意图着手编程，从而得到更加清晰的测试。你的断言将是意图的声明。相反，如果已经编写了排列和行为部分，那么断言部分就更像是基于特定实现的细节了。

10.5.2　示例程序优先，或至少第二

让我们运行一个简单的例子。我们需要一个针对GeoServer的测试：当用户不再被跟踪时，返回空的位置信息。

c9/18/GeoServerTest.cpp

```
TEST(AGeoServer, AnswersUnknownLocationWhenUserNoLongerTracked) {
   CHECK_TRUE(locationIsUnknown(aUser));
}
```

虽然我们并不知道如何实现验证输出结果的代码，但我们知道输出应该是什么样的，并加以描述。在测试集中，我们定义了一个用来处理默认和出错行为的函数。

c9/18/GeoServerTest.cpp

```
TEST_GROUP(AGeoServer) {
```

```
   // ...
➤  bool locationIsUnknown(const string& user) {
➤     return false;
➤  }
};
```

验证完测试失败之后，开始定义行为部分。

c9/19/GeoServerTest.cpp

```
TEST(AGeoServer, AnswersUnknownLocationWhenUserNoLongerTracked) {
➤  server.stopTracking(aUser);

   CHECK_TRUE(locationIsUnknown(aUser));
}
```

让测试名成为指导方针，这里提供一个排列：

c9/20/GeoServerTest.cpp

```
TEST(AGeoServer, AnswersUnknownLocationWhenUserNoLongerTracked) {
➤  server.track(aUser);

   server.stopTracking(aUser);

   CHECK_TRUE(locationIsUnknown(aUser));
}
```

最后，让故障成为指导方针，这里提供了函数`locationIsUnknown()`的实现。

c9/20/GeoServerTest.cpp

```
TEST_GROUP(AGeoServer) {
   // ...

➤  bool locationIsUnknown(const string& user) {
➤     auto location = server.locationOf(user);
➤
➤     return location.latitude() == numeric_limits<double>::infinity();
➤  }
};
```

“简直丑陋极了”，有人这么说。在Location类中增加一个功能用以判断一个位置是否是“未知的”，并修改`locationIsUnknown()`的实现。

c9/21/GeoServerTest.cpp

```
TEST_GROUP(AGeoServer) {
   // ...
➤  bool locationIsUnknown(const string& user) {
➤     return server.locationOf(user).isUnknown();
➤  }
};
```

我们马上意识到辅助函数已经不再发挥作用，于是我们彻底删除了它。

c9/22/GeoServerTest.cpp

```
TEST(AGeoServer, AnswersUnknownLocationWhenUserNoLongerTracked) {
   server.track(aUser);

   server.stopTracking(aUser);

   CHECK_TRUE(server.locationOf(aUser).isUnknown());
}
```

哦！重点是什么？或许从一开始就应该用这种方式设计Location类，并立即编写单行的断言。

也许是，也许不是。重要的是断言最终是一个简单的声明，而不在于它是如何实现的。你将经常需要密集、详细的代码语句来验证测试，即使不是从声明断言的用意开始的，你仍然需要从中抽取部分代码来放进一个辅助函数中。

需要多行声明的断言就留给读者实践了。

c9/23/GeoServerTest.cpp

```
TEST(AGeoServer, AnswersUnknownLocationWhenUserNoLongerTracked) {
   server.track(aUser);

   server.stopTracking(aUser);

   // 读取数据很慢，修复这个问题
   auto location = server.locationOf(aUser);
   CHECK_EQUAL(numeric_limits<double>::infinity(), location.latitude());
}
```

读者除了需要理解两行断言的步骤，还要决定它们之间的依赖关系（CHECK语句的参数引用了前一行返回的位置）。读者还需要思考如何将两行断言合成一个概念（即用户的位置信息是未知的）。

10.6 结束语

你在这一章学习了TDD的一些边边角角的知识。就TDD而言，这就是全部内容了！

不，这只是个玩笑而已，事情并非这样。本书提供的内容能让你开始深入挖掘TDD，但TDD的更多方面有待你去探索和发现。世界各地的测试驱动开发者正在尝试使用TPP，并想知道他们到底能在这条路上走多远。各地的行为驱动开发者（请通过google搜索BDD）正在寻找进一步消除商业和开发团队间差别的方法。验收测试驱动开发者正在努力尝试更好地理解TDD和ATDD之间的界限。

相对于现实存在的挑战（比如，如何让一个软件开发团队通过TDD获得成功），TDD的具体细节相对比较简单。一旦接受了TDD模式，你会希望其他开发者理解你对TDD的狂热。你也想在实践中不断提高。在下一章节中，你将学习如何可持续地发展TDD的策略和技巧。

第 11 章

发展和维持测试驱动开发

11.1 开场白

你已经学习了什么是TDD、它能做些什么、怎么做、为什么这样做以及何时使用它。如果持续践行TDD直到被其优势所征服，那么你迟早会变成测试控（test infected）[①]——你会坚持将它作为主要的编程工具。但是你很快就会意识到，并不是每个人都这么认为。即便身处应该使用TDD方法开发程序的团队，你还是会感受到程度不一的坚持和支持；也会遇到抵制与不屑，这通常是由缺少相关信息导致的。即便你的团队能克服最初的障碍让每个人都使用TDD，你依然会发现，保持热情不减是一大挑战——正如你的代码。维持TDD绝非易事。

本章将提供各式各样的主意来帮助你维系践行TDD的能力。你将学到以下内容：

- 怎样回答有关TDD的疑问和质疑；
- 一些得益于TDD的研究；
- 怎样避免“讨厌的测试死亡缠绕”；
- 怎样用结对编程维持测试驱动，并回顾为之付出的努力；
- 怎样更好地用katas和dojos践行TDD；
- 怎样避免测试覆盖度量的滥用；
- 持续集成是怎样成为TDD基石的；
- 有助于派生TDD标准的问题；
- 哪里可以获得更多TDD社区的实践信息。

11.2 向非技术人员解释测试驱动开发

或许你是个测试控，但你的热情还不足以改变别人的信仰，尤其是那些本身就不是程序员的人。本节将提供两个工具来支撑你与非测试控的对话。首先，针对一些常见的问题、故意营造的

① 参见“Test Infected: Programmers Love Writing Tests,”http://junit.sourceforge.net/doc/testinfected/testing.htm。

对话，提供一些简要、但具有强烈推销口吻的答案。其次，列举一些有效使用TDD的研究成果，它们将作为有力的论据来说服那些在倾听你说话前需要先调研的人。

11.2.1 测试驱动什么

提问：什么是TDD？

回答：它是程序员使用的一个软件开发技巧，用以增量地设计他们的系统。

提问：它是什么样子的？

回答：程序员将工作划分为小块的任务或单元。对于每一小块任务，他们写一个小的测试来说明代码应该怎样工作。然后再写一小段代码使测试通过。一旦测试通过，就要确保清理掉刚刚编写的小段代码中的所有设计或编码缺陷。

提问：他们只是编写单元测试？

回答：他们使用单元测试框架，以帮助确立新加的一段代码该怎样工作。是的，他们写单元测试，但这些测试同时被当作系统工作的文档。

提问：他们只是编写单元测试。

回答：他们将写单元测试作为增量设计系统的方法。测试是先写出来的。

提问：我没有看出这与单元测试的区别。先写测试会怎样？

回答：你会得到两个不同的结果。事后测试不会改变任何东西。事后测试或许能帮助程序员发现一些问题，然而仅仅是单元测试的话，则不足以迫使程序员改善设计和代码质量。

提问：TDD真的能改善设计和代码质量吗？

回答：有些研究（参见11.2.2节）表明，应用TDD会减少代码缺陷。另外，TDD的使用允许程序员持续地改善代码和设计。

提问：TDD怎么让程序员比以前更多地修改代码呢？

回答：程序员向系统中加入的每一小块代码都有一个表述其行为的测试。这意味着系统中的所有代码都经过了测试。所有的代码都有测试意味着，程序员可以做出保持代码整洁的小型改动。如果没有足够的测试，那现实就印证了那句格言：“如果没有问题，就不要修复。”

提问：那又怎么样呢？如果不必担心代码的整洁度，那不是也能节约时间吗？

回答：有研究表明，软件开发中80%的时间都用于维护（不是修复）软件[1]。随着时间的推移，代码库会变得很大，本来只需要花费几个小时就能完成的任务可能要花费几天。甚至一个简简单单的问题——“在这种情况下，软件做了什么？”——都可能需要程序员花好几个小时，扎进复杂的代码库中寻求答案。

提问：这是在说程序员素质的问题吗？难道他们不能一开始就写出整洁的代码吗？

[1] http://en.wikipedia.org/wiki/Software_maintenance

回答：编程好比写作。优秀的作家也需要不断地修改文章的语句和段落，提升可理解性。即便如此，他们依然会收到编辑审阅书稿时提出的许多问题。编写出能提供期望行为的代码就已经是一个挑战，所以第一步是把事情做对。第二步是改善代码，让其成为易于维护的设计的一部分。

提问：我还是不能理解为什么在代码完成后写一些单元测试不能达成同样的结果呢？

回答：由于人为因素，这不会发生。首先，一旦写好代码，大多数程序员就认为他们完成了“真正”的工作。他们自信满满地认为能写出正确的代码，经常满足于一两个简单的手动测试。因此，他们对额外写一些测试来证明他们已经知道的东西并不感兴趣。他们也会认为代码编写得足够好了，于是就很少利用测试整理代码。一般来说，程序员完成管理层的任务时会尽量少做。第二，日程安排压力通常起主导作用。任何事（代码编写完成）后的事情只会得到短期的关注。

提问：就哪些代码需要测试这一问题，难道不该让程序员做专业的判断吗？有没有代码简单到不需要任何级别的测试？坚持测试所有东西是不是有点浪费时间？

回答：大部分系统不会有太多简单到不会被破坏的代码，所以不会节省很多测试时间。我同样不会惊讶于程序员在看似完美无缺的系统中发现缺陷。代码缺陷会导致多种方式的开销，而且，用TDD减少缺陷也会降低开销。在了解系统如何工作这个看似简单的问题上也可以节省很多时间。

程序员在尝试为大型、已有的代码库编写单元测试时都会说，这很困难。主要原因在于，代码库没有按照可测试的思维组织，结果导致为这种系统写测试尤为困难。一般的程序员面对此挑战时会选择放弃。

提问：我听有些程序员说，实际上只要70%的代码有单元测试，其他部分可以交由功能测试。

回答：提供快速反馈的测试不能覆盖系统中30%的代码，不会令你如坐针毡吗？如果系统接近1/3的代码中确实有缺陷，那么只能在很久后才能发现它们。如果需要改动这部分代码以接纳新功能，这会是个相当慢长的过程。你需要增加单元测试（对遗留代码而言困难得多）或使用慢速的功能测试来保证系统不被破坏。

提问：为每个逻辑提供快速的测试是合理的。但最终会不会产生更多以测试形式存在的代码，而你又必须要维护它们？

回答：不会。大部分系统的代码量至少是其应有的两倍。部分原因是，程序员加入新代码时无法安全地移除重复代码。许多证据表明，单元测试的代码量可能和产品代码库相当，甚至稍微多一点。但是，完全使用TDD得来的代码库的代码量将减半。因此，代码量上扯平了，而且还收获了TDD的其他好处。

提问：我听说TDD不能真正抓到很多代码缺陷。这是不是意味着完全在浪费时间？

回答：TDD一开始就能避免将代码缺陷引入系统。你总是先写测试，使其通过，如果没能通过，你会在提交代码前先修复。这比先提交代码，然后在很晚之后才发现代码

缺陷要好得多。

提问：你让TDD听起来像个良方。

回答：TDD是个很棒的工具，但只是大小刚好的工具箱中的一个工具而已。光TDD是不够的。独立地验证小段代码并不能证明可以将它们组合在一起达成想要的功能。你还需要验收测试，可能还要包含性能测试、负载测试以及其他形式的集成测试，或许还有一定程度的探索性测试。

11.2.2 关于 TDD 的研究

有关TDD的有效性和成本已经有很多研究。下面这张表格总结自George Dinwiddie的维基页面[①]。

作者/年份	重要发现
Nagappan, N. et al, 2008	TDD消除缺陷率是40%~90%，其成本在开发初期会多出15%~35%的时间
Braithwaite, K., 2008[②]	TDD和代码复杂度成反比
Sanchez, J.C., et. al., 2007	TDD产生的缺陷密度低于行业标准。 TDD或许能减少代码复杂度随着软件年限而增长的程度
Bhat, T., 2006	TDD的使用可以让代码质量大幅提升，初始成本至少增加15%
Siniaalto, M. 2006	有些情况下，TDD会大幅提升生产效率，大概2/13的情况下会降低生产效率（但会提升代码质量）
Erdogmus, H. 2005	写更多测试的学生生产效率更高。至少代码质量和测试数目成线性关系
George, B. et al., 2003	测试驱动开发者生产高质量的代码（可以通过18%或更多的功能测试），且多用16%的开发时间。事后写测试的程序员写的测试不够充分

一个研究或许不具有说服力，但是大半的研究显示了相同的结果，这为以下两点提供了强有力的论据。

- TDD会开发出质量更高的代码。
- TDD会增加初始开发成本。

以下几点假设还没有相关的研究。

- TDD可以降低长期成本。
- TDD产生的测试会减少回答关于系统行为问题的时间。

继续使用TDD的大部分开发人员并不是基于这些研究结果才选择这一方法。相反，这是因为他们体会到了其带来的巨大益处，大部分人会告诉你使用TDD为他们的开发生涯带来了怎样的影响。

① http://biblio.gdinwiddie.com/biblio/StudiesOfTestDrivenDevelopment
② “Measuring the Effective of TDD on Design,” http://s3-eu-west-1.amazonaws.com/presentations2012/5_presentation.pdf

11.3　不良测试的死亡漩涡（亦称为 SCUMmy 周期）

有时团队开始使用TDD后，会在一段时期内得到良好的效果。而后事情开始慢慢被忽略，然后快速被忽略，最终决定放弃TDD。是什么导致了这个“讨厌的测试死亡漩涡”？怎样才能避免它呢？

这个问题不仅仅存在于TDD中。同样也有“讨厌的敏捷死亡漩涡”，伪敏捷的短迭代周期似乎在一段时间内产生了良好的效果。但1年或18个月后，团队会对手头上一堆混乱的场面十分吃惊。其结果就是敏捷方法被抛弃，背负着浪费时间的骂名。

Ben Rady和Rod Coffin在Agile2009峰会上名为“Continuous Testing Evolved”[①]的演讲中描述了SCUMmy周期。缩写SCUM描述了以下几种导致退化的不良测试的特征：慢速（slow）、令人疑惑（confusing）、不可靠（unreliable）、遗漏（missing）。下面是一个可能走向漩涡的情况（我在一些团队中看到过几次此类情境）。

(1) *团队写的测试大部分是集成测试*。这些测试和不稳定或慢速依赖紧密耦合，如数据库或其他外部API。虽然这会导致更慢的测试，但开始不会觉得有很大影响，因为仍可以在1~2分钟内运行完几百个测试。（想想“温水煮青蛙”。）

(2) *测试变多后带来的问题超越心理承受底线*。现在运行完测试需要好几分钟。

(3) *开发人员运行测试的频率变低，或者只运行一个测试子集*。同时，团队成员发现运行测试的问题增多。测试变得更长，需要更多的必要初始化，一旦出现问题，则需要更多的精力来进行理解和分析。其他问题也开始慢慢显现，由于依赖单元测试控制之外的不稳定因素，测试会间歇性地失败。开发人员发现测试经常上演“狼来了”，这意味着问题不是出在产品系统自身，而是测试设计。

(4) *开发人员删掉测试*。对于有问题的测试，本能反应就是禁用甚至删掉。开发人员发现，删掉测试比花费一个小时修复它们更容易。

(5) *代码缺陷开始变多*。剩下的测试很可能覆盖不了足够多的逻辑，在防止代码缺陷方面价值较小。（在文档化价值方面也大打折扣。）

(6) *团队或管理层质疑TDD的价值*。团队试图继续，但很明显这是徒劳的。

(7) *团队放弃TDD*。管理层记下这一明显的失败。

团队该怎么办呢？如果早知如此，那又何必当初呢？

理想状况下，你已经从本书中学到了TDD，它要求限制每个测试的范围，只测试一小段独立的逻辑。以这种方式构建的系统不太可能卷入不良测试的死亡漩涡。而且，不是用了TDD就能神奇地产生高质量的系统。你和团队必须不遗余力地去掉测试和产品代码中的不良设计。这同样要求团队知道良好的测试和代码是什么样子的。

---

① http://agile2009.agilealliance.org/files/session_pdfs/ContinuousTestingEvolved.pdf

下面的步骤将撤回迈入漩涡的步伐。如果你积极地观察发生的事情，就可能不会再次在漩涡中越陷越深。

(1) 团队写的测试大部分是集成测试。学习怎样编写单元测试。重读本书、参加培训、雇一个教练、举办Dojos、更多地审查、更多地阅读，等等。同时也要增加关于良好设计及代码结构的知识。

(2) 测试变多后带来的问题超越心理承受底线。将测试分为慢速和快速测试集。设立快测试的标准。（在一台开发机上需要5毫秒或更少时间？）如果团队成员向慢速测试集中加入一个测试，需要告知大家。学习重构测试及相关代码要做的事情，以便让测试变快。养成习惯，增量、但经常地尝试将慢速测试改进为快速测试。

(3) 开发人员运行测试的频率变低，或者只运行一个测试子集。如果测试运行超过慢速阈值，那么将测试集标为失败。（我最近为客户成功地改动Google Test做到了这点。）这样做会强化开发团队认识到快速测试的重要性。

(4) 开发人员删掉测试。监测测试覆盖率。虽然建立覆盖率目标的价值有待商榷（参见11.6节），你还是倾向于不断增加或至少有一个稳定的覆盖率数据。不幸的是，用良好的单元测试代替不良测试并获得同样的覆盖率可能要费点功夫。但在养成正确的习惯前，最好花点时间往对的方向走，而非放弃。

(5) 代码缺陷开始变多。对一个代码缺陷的首要任务是写一个测试。代码缺陷为认识TDD实践中的不足提供了机会。对于每个缺陷，要坚持在修复问题前写一个运行失败的单元测试。

(6) 团队或管理层质疑TDD的价值。精益求精。坚信TDD实践及其他实践（如验收测试、重构和结对编程）从长远来看主要致力于产出高质量的软件，而非仅仅减少代码缺陷。确保明白这些实践如何以及为什么和质量挂钩，同时确保团队成员以一种有助于达成高质量目标的方式践行它们。缺乏对质量的关注将导致系统开发失败——“不良测试的死亡漩涡”更不可饶恕。

(7) 团队放弃TDD。不要坐以待毙！管理层很少会容忍再次迈向他们认为必定失败的步伐。

和其他事情一样，你可能会因不当地使用TDD而导致失败。但也有可能成功，并且是大获成功，否则的话，我也不会大动干戈地写下本书。由于未能正确地坚持一项技术而认为它本身不好，可是非常不好的！

11.4 结对编程

你写的测试和产品代码等同于高昂的投资。理想情况下，你会假设所有的开发者都经历了必要的培训，更重要的是，要有高效地编写高质量代码的意愿。但只要看一下任何现存的系统就知道并不是那么回事。

在本书中，我已经尝试让大部分系统一团糟，这是因为程序员没有有效保障代码不会退化的机制（如TDD）。当然也有很多其他原因。

- **缺乏教育**：太多的程序员不理解核心的设计原则和良好的编程结构。其中一些人认为他

们知道的够多了，不承认还有更多的东西要学。

- **缺乏担忧**：太多的程序员不在乎编写出难懂的代码。或者他们会为此辩护，说系统中已经充满了不良代码。
- **时间压力**：你听到过多少次"发布吧——我们没有时间关心质量了"？
- **缺乏审查**：大多数是因为一个捣蛋的程序员往代码库中加入了具有严重破坏性的代码。有时，这个程序员是一个身价昂贵的短期顾问。有时，在发现这些糟糕代码前，作者就已经不见踪影了，只剩下你独自面对这不一致且难以维护的代码。
- **缺乏合作**：在多于一个开发者的团队或一个大小合理的代码库中，编码风格的不同和代码质量的好坏会很快展现出来。在缺乏解释的情况下，理解别人写的一段代码可能非常困难，其他开发者或许会编写一个解决方案来处理已经解决了的问题。单个程序员可能在没有寻求别人意见的情况下编写出不够优化的代码。

11.4.1　结对原则

结对编程技术的意义在于，通过提供持续审查、合作及来自队友的压力来帮助开发程序。下面总结了结对编程是怎样工作的。

- 两个程序员积极地共同开发一个解决方案。
- 程序员通常肩并肩地坐在一起（虽然包括远程结对在内的其他方式也是有可能的）。
- 在任意给定时刻，每个人扮演以下角色之一：*驱动*，即积极地编写解决方案；*领航*，即提供审查和策略指导。结对*不是*一个人做，另一个人坐在后面看。
- 结对的程序员要在结对期间经常互换角色，可以在测试失败或成功时互换。
- 结对是暂时的，每90分钟就重新结对。轮流结对的主要目的是团队内部间可以增加知识，相应地降低风险。

和TDD很像，结对编程的相关研究[①]表明，代码质量提升的同时开发初期成本也会增加。你应该想起"人多智广"这句格言了。可能和你开始想的一样，成本不会成倍增长。积极的代码审查带来的高质量是一个原因。来自队友的压力和沟通程度的提高、协作以及教导也是带来更低成本的原因。其他好处也很多，例如，降低了风险、提升了灵活度[②]。

11.4.2　结对编程与测试驱动开发

TDD和结对编程是天作之合。如果有一个支持体制的话，那么学习TDD会变得非常容易。即使队友不施加任何压力，开发者也很可能丢弃旧的、非TDD的习惯。同有经验的测试驱动开发者

---

① 参见Dybå, T. et. al. "Are Two Heads Better Than One? On the Effectiveness of Pair Programming," at http://dl.acm.org/citation.cfm?id=1309094, 这总结了一些关于结对编程的研究。

② 参见http://pragprog.com/magazines/2011-07/pair-programming-benefits来了解更多的潜在好处。

并肩作战会让习得TDD的过程事半功倍。交换伙伴有助于用心地写好测试。

TDD周期也为结对期间的角色变换提供了天然的切换点。许多程序员使用乒乓结对。第一个程序员一直写测试代码，直到失败，然后将键盘交给第二个程序员。第二个程序员编写产品代码，让测试通过，然后接着写测试的其余部分或下一个测试。一旦测试通过，队友间可以在各类重构过程中互换角色。

队友间会时不时地讨论测试方向，特别是在他们了解彼此偏好风格的情况下。一般的经验是，在队友拿到键盘并演示这些意图前，争论不要超过5分钟。通常情况下，这样得来的测试方向胜于经过讨论得来的。

11.4.3 角色切换

如前文所说，结对是暂时的。然而，比较自然的倾向是允许结对时间足够长，以便在这期间能完成一个任务或者整个功能。

当然，伙伴交换会带来*上下文切换*的开销。如果你新加入一个结对（*新手*），必须放弃对刚刚要解决的问题的深入思考。（我将你要结对的程序员称之为*内行*。）对大多数人而言，这没什么大不了的，打断5或10分钟也不算什么。但随后要面临更难跟上新问题的挑战。根据问题的难度，可能只需要几分钟，但也可能需要来自内行详细且耗时的解释。

如果采用TDD，关注点就不同了。相比于来自内行的详细解释，你会更专注于当前失败的测试。（如果没有的话，那就在内行添加时边听边看好了。）此时的目标是阅读测试名称并确保其是合理的。然后可以阅读测试来帮助理解当前的编程任务。

只关注单个测试，特别是编写良好、目的精确的单元测试，能够让你更容易地开始作出贡献。你不需要了解所有细节。相比于提前列出详细冗长的介绍，内行会慢慢地指导你前进。

团队在任务间切换得越多，就变得越容易，特别是在一个小团队里。但为什么不在任务完成前防止切换呢？

记住，结对的一个关键目的是代码审查。通常而言，人多智广，但结对依然有可能导致不良的生产力。这很可能会发生在结对深入解决方案，并到达一个他们可以说服自己只有一个正确答案的点时。此时，如果一个没有深入解决方案的局外人和结对分享一些思考，有利于从全新的视角切入从而避免不良的解决方案。

此外，加强结对互换会提升新加入的队友熟悉特定部分代码的可能性。在大小适中的系统中定期地结对互换，可以让每个人最终了解系统的全部。

如果团队努力坚持的话，结对互换的另一个重要益处是代码质量的提升。因为质量更好的代码会减少新人理解代码的时间，这有助于降低上下文切换的成本。必须要修改不清晰的代码。结对应该迅速知道为了清晰而持续重构可以节省时间，对测试更是如此。

结对互换体现了增加的初期成本和长期收益间的权衡，和结对编程自身、TDD以及许多敏捷软件开发特点非常类似。上下文切换当然会带来短期的痛苦，但随着时间的推移，你能够学习到怎样减轻痛苦。然而，在满是孤立开发者的团队中，由少量代码审查及没有分享知识造成的痛苦会伴随时间一直持续下去。

11.5　Kata 和 Dojo

除了持续地学习新知识，成功的专业人士会定期锻炼他们的手艺。音乐家将音阶作为热身训练、整形外科医生用尸体做练习、运动员做一些日常操练和训练比赛、演说家会对着镜子做热身、武术家练习Kata——精心组织的动作模式。

在这些练习时段里，练习者会重复其职业生涯初期习得的常见和基本的要素。这些操练有助于加深基本技能。同时，练习者在表演或比赛前也会用这些做热身。在比赛期间，经验丰富的人会尽可能地凭靠“肌肉记忆”处理基本的动作。这样有益于提升他们的思考能力，从而对其所处的环境作出更好的反应。

此外，对基本动作的掌握是进一步成长和探索的基本功。更加复杂和有效的招式常常是基本招式的创新变种。高级专业人士有时甚至会发现，他们可以忘掉基本的规则。他们对基本招式的精深把控，使其意识到并接受忘却它们带来的成本。[①]

许多开发者将武术中Kata的概念应用到TDD中。两者的理念类似：测试驱动开发出对应简单编程问题的解决方案。不断重复这个训练，直到可以展示通往解决方案的理想路径，在这个路径上没有多余的步骤。Dave Thomas在他的个人站点上提供了一些示例问题（http://codekata.pragprog.com）。

11.5.1　在测试驱动开发中应用 Kata

Kata对于练习TDD也奏效吗？为了构建软件，在任何特定时刻你都要和大量的工具打交道。你的双手要用键盘编写代码；你要同一个编辑器交互，它提供了完成任务的许多方法；你也要使用一门语言及一些程序库（包括一个测试框架），同样，它们也提供了达成解决方案的无数方法。最终，你要思考很多东西。虽然打造解决方案的过程貌似是唯一、不可重复的，但在软件开发中依然有许多重复性的主题（模式）。

越多地练习这些要素，就越能更好地掌握它们。最难的部分无疑是思考，但依然可以通过解决编程问题来更好地思考。

从哪里开始呢？找到一个感兴趣的Kata，或许就是之前所演示的那个。对于第一个Kata，最

① 在武术中有句话——无招胜有招。其意思是基本的招式已经固化为本能，习武者会本能地做出一些应对招式，而不需要经过大脑思考。忘掉基本招式反而使反应更快，因为思考的成本更高。——译者注

好找一个能在一个小时内完成的小问题。记下时间，然后开始测试驱动开发解决方案。如果被问题卡住，要甘愿后退一点点。最后记录下完成解决方案所需要的时间。

如果你绝望地深陷泥潭，那么就彻底放弃，重新尝试，必要时寻求帮助。（你将知道是否需要选择另外的Kata作为第一次练习。）

审查你的解决方案，思考怎么得出这个解决方案的。再一次练习这个Kata，最好是马上或者在一段合理的时间内，以便记住刚学习到的东西。不要去记忆，而是要思考走好下一步。与第一次练习花费的时间进行对比。

时不时地重复这个Kata，大概一周1~3次。找出需要改进和改善的地方。例如，寻求更好的API或语言结构，并使用更好的编辑器快捷键以减少敲键盘的次数。你将在某个点接近理想的解决方案，并具备以最少的步骤和错误得出此方案的能力。现在它成了你热身练习包中的一个工具。

要想提高测试驱动解决问题的能力，需要丰富热身练习包。解决不同类型的问题能够教会你不同的技巧，有助于获得更重要的洞察力。

附录B中包含了一个简单的出发点——罗马转换器练习，在这个练习中，你要测试驱动开发出一个算法，用来将阿拉伯数字转换为对应的罗马数字。你也可以尝试来自本书的其他练习，例如，将Soundex作为你的第一个Kata。表1中描述了许多有用的Kata资源。

表1 Kata资源

站 点 名	网 址	描 述
CodeKata	http://codekata.prag-prog.com	Dave Thomas最初的Kata站点
Craftmanship Katalogue	http://craftsman-ship.sv.cmu.edu/katas	各种Kata的评分和资源
Coders Dojo	http://codersdojo.org	允许尝试并分享Kata（Ruby语言版本）
Software Craftmanship Code Kata	http://katas.softwarecrafts-manship.org	包含各式Kata的视频
TDD Problems	https://sites.google.com/site/tddproblems/	主要为演示测试驱动而设计，但也可以用作Kata
cyber-dojo	http://www.cyber-dojo.com	一个在线的Dojo，允许在浏览器中编码和测试

11.5.2 Dojo

Kata是典型的单人训练（虽然也可很好地用于结对训练）。你可以随时练习，并在5分钟或15分钟后停止。在训练场中，以团队的形式练习Kata可以更进一步地掌握TDD。类似于武术中的Dojo，测试驱动开发Dojo是群组训练，其中有一些级别的仪式和结构。

对于典型的Dojo，将时间定为60~90分钟，找一个有投影的房间，让每个人都能看到屏幕。为了让团队对Dojo的概念有所认识，你的第一次Dojo练习可能需要一个或一对演示人员，由他们示范怎样对一个Kata演绎出最终的解决方案，且他们事先已经完成过这个Kata。演示人员的工作是确保所有人理解他们在进行下一步前的编程选择。

对之后的Dojo来说，randori-style更具娱乐性和带入性。有很多各式各样的randori dojo结构；下面是一个在Coding Dojo wiki（http://codingdojo.org）上的描述。

- 开始进行Dojo时，由结对处理整个组选择的问题。
- 结对中的程序员在开发者和指引者间频繁切换角色，可以使用乒乓结对。
- 这个结对有大概5~15分钟的时间限制，用来推进解决方案。开发时，他们应该描述正在做的工作。
- 到限制时间时，结对中的一个成员（通常是开发者）被换出，由观众成员代替。所有的Dojo参与者在整个过程中至少被换入结对一次。
- 观众成员可以适当地提出建议。

你可以选择一个老师，通过提问（但不给出答案）的方式来给你提供指导建议，并在其他方面做些工作来辅助Dojo训练。

如果Kata进行超过半个小时，不妨中途休息5分钟。结束时做一个简要的回顾：讨论整个过程中做的好的地方，决定下一次要采用的不同方式。

Randori Dojo为团队的每个人提供了参与的机会。当团队要处理新问题或某个开发者要演示他认为更有效的解决方案时，你或许可以切换回演示风格。

Dojo的要点在于合作与分享。团队可以更好地理解别人是怎样解决问题的，你也会在这个过程中学到许多新点子。

11.6　有效地使用代码覆盖率统计

代码覆盖率——单元测试覆盖的代码行比例——是一个新的“代码行”统计方法，肯定会被许多懒惰的经理滥用。对此统计方法最幼稚的理解是：100%意味着代码被全面覆盖，而0%表示你甚至没做过这方面的努力。你可以找到很多C++代码覆盖率工具，大部分是收费的。COVTOOL（http://covtool.sourceforge.net）是一个开源的测试覆盖率工具。

好的覆盖率工具也会“注释”代码，在运行测试时可以标明哪些特定的代码行会被执行。度量覆盖率的真正价值在于，知道哪些代码没有被覆盖有助于决定哪里需要添加测试。

如果一直遵循简单的TDD周期来开发代码，代码覆盖率将逼近100%。（代码覆盖率或许永远不会到达100%，这受限于工具自身，以及在识别特定代码怎样被使用方面的功能。这没什么问题。）你已经知道了这个情况——如果只在测试失败的情况下写代码，并且所写代码量不多于让失败测试通过，从定义来看，就达到了100%的覆盖率。就个人而言，我不会因此而担忧代码测试工具。但如果你坚持先写测试，或者倾向写多于让测试通过的代码量，代码覆盖率工具能够提供有价值的反馈。

系统范围的平均覆盖率低于90%的话，说明代码库不是（不完全或根本没有）用TDD开发的，

仅此而已。覆盖率达到90%或更多，传达的信息也很有限——通常意味着代码库是用TDD构建的，但使用集成测试覆盖大量代码也是有可能的。更典型的是，未用TDD构建的系统的整体覆盖率非常低（以我的经验，很少高于70%）。集成测试很难覆盖每个末分支，同样，在代码编写完后很难编写全面的单元测试。

用TDD编写精准的单元测试也有可能达到90%以上的覆盖率，但创建的测试却难以维护。换言之，高覆盖率可能看上去很好，但却没有说明所有的信息。

永远不要将代码覆盖率作为目标。坚决主张高覆盖率的经理会得到诉求的答案——高的覆盖率数据，其他用途寥寥无几。

11.7　持续集成

你在职业生涯中听过多少次“在我的机器上是工作的”这样的话？或者你自己也说过？除非你的代码集成了其他的产品代码，并且在基准机器（这台机器上的环境和生产环境类似）上运行了所有的测试，否则你不会真正得知它是否工作。（直到在真正的客户那运行成功了，你才能知道它真的可以工作）

持续集成（continuous integration，CI）服务器的工作是检测源码仓库，在提交代码时开启构建流程。一旦构建完成，持续集成服务器通知所有对此事件感兴趣的人，保留构建输出以便日后查看。

构建脚本应该编译、链接、部署、运行单元测试及其他测试，做任何你认为必要的事情，证明系统可以发布（至少接近发布）。一些能力很强的团队有足够的信心，认为他们的构建已经演化到持续发布的级别——每次构建成功就可以部署到生产环境。

持续集成的构建可能会花一些时间，但也还好。拥有持续集成服务器可以让你持续工作，一旦它发现问题，你就能及时知道。但要小心，不要因为有了工具就满足于总体构建时间。缓慢、耗时不断变长的构建过程很快会成为问题。一天得到几次反馈是必要的。

在TDD环境中，持续集成服务器是基础工具。没有理由不拥有一个。测试（包括所有的非单元测试）是代表整体系统健康的最好晴雨表。提交代码时，你想要知道代码给系统带来的价值，以及代码不会破坏最新代码库中的任何功能。

确保你的团队对基于持续集成的所有流程意见一致（参见11.8节）。你应该知道提交代码后会发生什么，也应该知道持续集成服务器报告构建错误时会发生什么。

有许多持续集成服务器供你使用。知名的工具包括：Jenkins、buildbot、CruiseControl和TeamCity。要根据环境及构建脚本选择最适合的。在StackOverflow上可以找到有关合适的持续集成服务器的讨论[1]。

---

① 参见http://stackoverflow.com/questions/145586/what-continuous-integration-tool-is-best-for-a-c-project。

11.8　为团队制定测试驱动开发标准

要确保团队在几个简单的标准下使用TDD方法。不要让这些标准成为起步的绊脚石。正如敏捷和TDD的其他东西一样，其目标是采取小步伐，然后在前行的过程中持续改善。

下面是需要认同的一些关键事情。

- 单元测试工具（如Google Test、CppUTest、CppUnit）。随着时间的推移，你可以使用以后出现的更好的工具。那时，增量地迁移是可以的。但就目前而言，如果没有更好的理由不使用它，那么你应该找到最适合团队的单元测试工具，并坚持使用。
- 其他工具，包括模拟框架和代码覆盖率工具。
- 集成标准。在集成代码进入源代码管理系统前，团队成员应该对采取的测试级别达成共识。理想情况下，开发人员应该运行所有的单元测试，前提是它不会成为频繁集成的障碍（因为遗留系统产生的问题）。如果有测试失败，那么要杜绝提交代码。
- 测试运行标准。测试怎么样算是慢的？测试可以向控制台输出信息吗（最好不要）？
- 失败流程。当构建失败，应该发生什么，哪些人要参与进来？
- 禁用、注释掉测试。一般来说，坚持不要将这样的改动提交进代码库。如果有充足的原因，代码提交者应该提供解释性注释。
- 测试名称格式。DoesSomethingWhenSomeContextExists是个好的出发点，但要避免在此方面犯教条主义。准确性和可读性是重要的因素。
- 测试结构。遵从了AAA吗？怎样命名fixture的？测试的物理布局是怎样的（从包和文件组织的角度出发）？
- 断言格式。是Hamcrest，还是其他的？断言的注释可行吗？

在会议中花一个小时讨论代码库的标准、记录遇到的问题、达成共识，然后继续下去。如果认识到标准与代码冲突，那么就修改标准。拥有标准虽好，但绝不要让它成为前进的拦路石。

11.9　保持与社区同步

TDD是永无止境的探索之旅。不断有新的、更好的想法来更好地实践TDD。使用TDD这数十年来，我个人不断地汲取关于它的新主意。我在2000年写下的测试和代码让我感到失望，甚至昨天写的测试和代码同样让我失望。这没关系，因为TDD周期是构建在持续改善的理念之上。

本节将提供一些主意，让你可以去TDD社区找到新的、有趣的东西。

11.9.1　阅读测试

学会怎样阅读代码是值得获取的技能。更好的阅读代码方法是先观察描述其行为的测试。

开源项目为了解别人使用TDD的方式提供了捷径。你会发现大家的风格多种多样，毫无疑问，你已经可以识别阅读测试中不好的实践。

不要只局限于阅读C++测试。TDD的核心原则同样适用于其他编程语言。

11.9.2 博客与论坛

参与讨论！下述表格给出了几个经常或一直讨论TDD的活跃论坛和博客：

站 点 名	URL	谁/什么
Test-Driven Development Yahoo!Group	http://tech.groups.yahoo.com/group/testdrivendevelopment/	基于邮件的论坛
Extreme Programming Yahoo!Group	http://tech.groups.yahoo.com/group/extremeprogramming/	基于邮件的论坛
LinkedIn Test-Driven Development Group	http://www.linkedin.com/groups/Test-Driven-Develop- ment-155678	基于网页的论坛
Agile Otter Blog	http://agileotter.blogspot.com/	Tim Ottinger, *Agile In a Flash*的合著者（*Agile in a Flash: Speed-Learning Agile Software Development[LO11]*）
James Grenning's Blog	http://www.renaissancesoft-ware.net/blog/	*Test-Driven Development in Embedded C*的合著者（*Embedded Test Driven Development Cycle[Gre07]*）
Michael Feathers	http://michaelfeathers.type-pad.com/michael_feath-ers_blog/	《修改代码的艺术》的作者（《修改代码的艺术》）
Uncle Bob	http://blog.8thlight.com/uncle-bob/archive.html	*Clean Code*的作者（*Clean Code: A Handbook of Agile Software Craftsmanship [Mar08]*）
Coding Is Like Cooking	http://emily-bache.blogspot.com/	Emily Bache，*The Coding Dojo Handbook*的作者（*The Coding Dojo Handbook [Bac12]*）
Sustainable Test-Driven Developement	http://www.sustainablet-dd.com/	Scott Bain和Amir Kolsky
Jeff's Blog	http://langrsoft.com/jeff/	我本人的

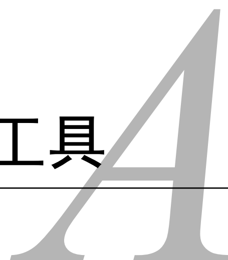
11.10 结束语

恭喜！你已经学习了TDD的大量知识，以及它可以怎样让你和你的团队受益。在本章中，你学到了让个人和团队保持热情不减的方法。

TDD的周期很简单：制定、构建、改善。本书已经提供了实践TDD的规范。必须通过持续地使用和练习来打造你自己的TDD知识库。最重要的是，你和你的团队要在本书的基础上不断地进取和改善。

附录 A

比较单元测试工具



A1.1　开场白

本书中的例子用到了Google Test、Google Mock和CppUTest。你可能已经使用了其他的单元测试工具，或者你的开发环境需要另一种不同的工具。在本章中，你将快速浏览一些特性以考虑为TDD选择合适的单元测试工具。

A1.2　测试驱动开发单元测试工具的特性

几乎任何你关心的工具都能满足TDD的需求。有些工具使其变得更简单，有些则使TDD变得困难而琐碎。很多工具都包含花里胡哨的附属功能，但可能永远不会在TDD中应用。有些工具有不同的设计考虑，比如最小内存占用，因此可能包含了一些与TDD的目标相冲突的特性。

在我看来，下列特性是测试驱动开发时必需的。

特　　性	说　　明
独立的测试名称	每个测试都能被唯一标识（倾向于使用“范围、集合、测试”的组合名称）。至少有一个工具支持事后为测试添加名称，默认情况下仅用数字表示测试
易于增加新的测试	测试应该自动注册，以便默认执行。工具不应该要求程序员分别注册定义好的测试。像CppUnit的一些旧工具需要程序员显示地将新增的测试注册到测试集中
支持fixture	支持定义提供设置、回收函数的fixture，将通用的辅助函数集中起来
独立的测试	每一个单元测试都能独立运行，不依赖于任何其他单元测试的输出
判断相等性	比较两个数量是否相等，提供清晰、有表现力的出错信息
测试套件	能运行任意单元测试的集合
支持mock	提供一套简易定义和使用mock的方法，或者提供对第三方mocking工具的链接支持

下列的特性并非必需，但可以给TDD提供帮助。

特　　性	说　　明
Hamcrest	提供增强的断言功能以支持匹配器（并支持自定义匹配器）。维持TDD要求测试具有高度可读性，Hamcrest有助于实现这个目标

（续）

特　　性	说　　明
定制化输出	提供定制化输出的功能。默认情况下，工具应该提供一个显示测试失败细节的简要总结
对异常的断言	提供一种直接的方法以断言抛出的异常
测试的统计结果	提供测试运行的统计信息，以及每个测试执行的次数
内存泄漏测试	提供一种可配置的管理机制来监测内存泄漏，比如一个测试申请了内存却没有释放
健壮性	即使有些测试抛出异常或者应用程序崩溃，测试集依然可以完成运行（在持续集成服务器上运行测试时，这个特性尤为重要）

下列特性虽然看上去不错，但和TDD没有关系。

特　　性	说　　明
参数化的测试	你可能会发现：在某些罕见情形下，向一个测试发送很多数据是非常有用的。然而测试驱动并非如此，且即使需要独立编写代码，那也是相当简单的
依赖性	对集成测试而言，让其按照指定顺序执行是非常有用的。但创建对其他测试结果有依赖关系的测试，对TDD来说毫无意义

A1.3　Google Mock 的说明

Google Mock为mock提供了内嵌的支持，还提供了一个应用广泛的Hamcrest库。

Google Mock不完全支持测试套件。它支持通过命令行过滤的方式运行一部分测试，但对固定不变的测试套件不提供支持（可以重定向输入为一组文件，绕开这个缺陷）。

遗憾的是，默认情况下Google Mock的输出结果包含了所有的测试信息。对于大的测试套件来说，如果不借助命令行操作，要想从大量的测试输出中找出失败测试的信息将会异常困难。

为了解决这个问题，可以创建一个自定义的测试事件监听程序，每隔几分钟就输出简化后的Google Test执行信息。更多信息请参见https://code.google.com/p/googletest/wiki/AdvancedGuide#Extending_Google_Test_by_Handling_Test_Events

A1.4　CppUTest 的说明

CppUTest提供了对TDD单元测试工具来说非常重要的绝大多数特性，同时也支持通过命令行切换的方式运行测试的一个子集，和Google Test的过滤器非常类似，但没有那么稳定。

CppUTest还具有一个非常重要的特性——支持内存泄漏监测，这个特性可以和其他单元测试工具结合起来使用。

A1.5 其他单元测试框架

你可能考虑到了下表列出的一些单元测试框架。

单元测试框架	链　　接
Boost.Test	http://www.boost.org/doc/libs/1_53_0/libs/test/doc/html/index.html
CppUnit	http://cppunit.sourceforge.net/doc/1.11.6/cppunit_cookbook.html
CppUnitLite	http://c2.com/cgi/wiki?CppUnitLite
CUTE	http://cute-test.com/
CxxTest	http://cxxtest.com/
Unit++	http://unitpp.sourceforge.net/

A1.6 结束语

C++单元测试工具在继续演化。这个附录的目的是帮助你制定一套准则，用以选择团队所需的单元测试开发工具，而不是总结截止到本书出版为止的所有单元测试工具。有些网站提供了可用工具的对比。http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle中的文章详细比较了几个工具，请注意，这篇文章写于2010年，没有讨论到CppUTest和Google Test。

一般来说，对适合的C++单元测试工具的选择不会大错特错。你的工具会随着时间越来越好，即便不是这样，你还有机会定制它。因此，选择一个工具，开始你的旅程吧。

附录 B

代码Kata：罗马数字转换器



B.1 开场白

在11.5节中，你了解了使用Katas有助于加固基本的TDD概念，特别是构建解决方案的增量方法。本附录提供了一个Kata供训练之用。通过一个个测试，你将测试开发出罗马数字转换器的实现。

B.2 出发吧

我们要构建什么？

场景：罗马数字转换器

我们需要一个函数，它接受一个阿拉伯数字，返回一个与之对应的罗马数字（以字符串的形式）。

B.2.1 1，2，3……

通过第一个测试要花费几分钟，因为要做一定量的初始化工作（创建构建脚本、加上头文件和包含语句，等等）。也要作一些决定：测试方法的名称是什么？函数接口应该是什么样子的？

我们将转换器作为自由函数。下面是第一个失败的测试：

roman/1/RomanConverterTest.cpp

```
TEST(RomanConverter, CanConvertPositiveDigits) {
     EXPECT_THAT(convert(1), Eq("I"));
}
```

Kata的一个目的是最小化我们的步伐。虽然我不喜欢自由函数，但如果只是纯粹的功能需求，它们确实是很好的开始。一旦功能完成，必要时可以将其变为类静态成员函数。

为了让事情变得更简单，暂时先将测试和`convert()`实现放在同一个文件中。下面是整个文

件的内容，其中包括能让第一个测试通过的代码。

roman/2/RomanConverterTest.cpp

```
#include "gmock/gmock.h"

using namespace ::testing;
using namespace std;

string convert(unsigned int arabic)
{
   return "I";
}

TEST(RomanConverter, CanConvertPositiveDigits) {
   EXPECT_THAT(convert(1), Eq("I"));
}
```

继续向前推进。转换数字2似乎是最适合的下一步。

roman/3/RomanConverterTest.cpp

```
TEST(RomanConverter, CanConvertPositiveDigits) {
   EXPECT_THAT(convert(1), Eq("I"));
   EXPECT_THAT(convert(2), Eq("II"));
}
```

你或许注意到了上面用的是EXPECT_THAT，而非我所喜爱的ASSERT_THAT。提示一下它们的区别，如果EXPECT_THAT断言失败，Google Test将继续运行当前测试。但如果ASSERT_THAT断言失败，Google Test将停止执行当前测试。

通常而言，你想要为每一个测试函数准备一个场景——一个单独的测试用例。在单独的测试用例中，在断言失败后继续运行测试意义不大，因为余下的测试逻辑通常也会失效。

就罗马数字转换器而言，创建新的测试方法意义不大，因为外部表现的行为（转换一个数字）在新的测试用例中没有改变。如果对两个测试用例使用不同的方法，函数名会变得乏味而且没什么价值：Convert1、Convert2。

因为一个测试函数有许多测试用例，所以使用EXPECT_THAT更加合理。如果一个断言失败了，我们仍然想知道其他的测试用例通过了还是失败了。

为了让第二个断言通过，这里引入if语句，将数字2的转换看作一个新的情况。这是我们到目前为止知道的全部内容。我们有两个测试用例——数字1和数字2——我们写的代码反映了我们的知识。

roman/3/RomanConverterTest.cpp

```
string convert(unsigned int arabic)
{
   if (arabic == 2)
      return "II";
```

```
   return "I";
}
```

下面是第三个断言：

roman/4/RomanConverterTest.cpp

```
   EXPECT_THAT(convert(3), Eq("III"));
}
```

这时我们察觉到了一个模式，可添加简单的循环来使用它。

roman/4/RomanConverterTest.cpp

```
string convert(unsigned int arabic)
{
   string roman{""};
   while (arabic-- > 0)
      roman += "I";
   return roman;
}
```

简单，简单。我们还没考虑优化。因此，就目前来说，能连成字符串已经够好了，`for`语句似乎是不必要的。

B.2.2　数字 10，先生！

1，2，3……准备好4了吗？没有必要。TDD并不强迫以特定的顺序构建测试。相反，我们应该思考一番，下一个测试应该是什么。

罗马字符IV（4）是一个减法，即比数字V（5）小I（1）。我们还没有考虑数字5（V），或许至少在处理数字5后，再回头看看数字4。至于V，它似乎是一个简单的特例。或许应该先思考一下到目前为止哪些是类似的。1、2、3的结果是递增的I、II、III。来看另一个类似的递增情况：X、XX、XXX。

roman/5/RomanConverterTest.cpp

```
EXPECT_THAT(convert(10), Eq("X"));
```

数字10是另一个特例，最适合用`if`语句处理。

roman/5/RomanConverterTest.cpp

```
string convert(unsigned int arabic)
{
   string roman{""};
   if (arabic == 10)
      return "X";
   while (arabic-- > 0)
      roman += "I";
   return roman;
}
```

数字11应该更有趣。

roman/6/RomanConverterTest.cpp

```
EXPECT_THAT(convert(11), Eq("XI"));
```

roman/6/RomanConverterTest.cpp

```
string convert(unsigned int arabic)
{
   string roman{""};
   if (arabic >= 10)
   {
      roman += "X";
      arabic -= 10;
   }
   while (arabic-- > 0)
      roman += "I";
   return roman;
}
```

如果转换器接受了一个大于10的数字，追加一个X，从数字中减去10，继续执行相同的`while`循环。数字12和13的断言应该可以自动通过。

roman/6/RomanConverterTest.cpp

```
EXPECT_THAT(convert(12), Eq("XII"));
EXPECT_THAT(convert(13), Eq("XIII"));
```

两个断言确实通过了。数字20的断言却失败了。

roman/7/RomanConverterTest.cpp

```
EXPECT_THAT(convert(20), Eq("XX"));
```

只要将关键字`if`改为`while`就可以通过测试了。

roman/7/RomanConverterTest.cpp

```
string convert(unsigned int arabic)
{
   string roman{""};
   while (arabic >= 10)
   {
      roman += "X";
      arabic -= 10;
   }
   while (arabic-- > 0)
      roman += "I";
   return roman;
}
```

B.2.3　欲擒故纵

我们获得了一个似乎有重复代码的实现。while循环有点类似。面对“几乎重复”，下一步是让它们变得更加相像。

roman/8/RomanConverterTest.cpp

```
string convert(unsigned int arabic)
{
   string roman{""};
   while (arabic >= 10)
   {
      roman += "X";
      arabic -= 10;
   }
   while (arabic >= 1)
   {
      roman += "I";
      arabic -= 1;
   }
   return roman;
}
```

重构后，上述两个循环有相同的逻辑，只是数据不一样。我们可以提取公有函数来杜绝重复代码，但这似乎不是高明的选择，因为两个分隔元素会变化（阿拉伯数字的和与罗马数字符串）。现在来提取转换表吧。

roman/9/RomanConverterTest.cpp

```
string convert(unsigned int arabic)
{
   const auto arabicToRomanConversions = {
      make_pair(10u, "X"),
      make_pair(1u, "I") };
   string roman{""};
   for (auto arabicToRoman: arabicToRomanConversions)
      while (arabic >= arabicToRoman.first)
      {
         roman += arabicToRoman.second;
         arabic -= arabicToRoman.first;
      }
   return roman;
}
```

这个重复的循环变成了通用的循环，成为了遍历转换表的一部分。

B.2.4　收尾工作

我们所说的算法是这样的：对于每一对阿拉伯数字与罗马数字映射，减去阿拉伯数值，然后追加相应的罗马数值，直到余数小于阿拉伯数字。（可以使用不同的数据结构，但要确保转换映

射按从大到小顺序遍历。）

细想一下就会发现，算法能够应对所有的数字映射，只需要将其加进表中。我们来试一下之前当成特例考虑的数字5。

roman/10/RomanConverterTest.cpp

```
EXPECT_THAT(convert(5), Eq("V"));
```

只需要向转换表中加入一项，测试就会通过。

roman/10/RomanConverterTest.cpp

```
const auto arabicToRomanConversions = {
   make_pair(10u, "X"),
➤  make_pair(5u, "V"),
   make_pair(1u, "I") };
```

现在写测试主要是为了增强信心（参见3.5节），因为除了向转换表中加入数据，我们什么也没做，但是这就可以了。再多一些这类测试吧！50、100，或者组合起来？

roman/11/RomanConverterTest.cpp

```
EXPECT_THAT(convert(50), Eq("L"));
EXPECT_THAT(convert(80), Eq("LXXX"));
EXPECT_THAT(convert(100), Eq("C"));
EXPECT_THAT(convert(288), Eq("CCLXXXVIII"));
```

我们发现，这些测试也很容易通过，只需要向转换表中加入必要的转换项即可。

roman/11/RomanConverterTest.cpp

```
const auto arabicToRomanConversions = {
   make_pair(100u, "C"),
   make_pair(50u, "L"),
   make_pair(10u, "X"),
   make_pair(5u, "V"),
   make_pair(1u, "I") };
```

最后，再一次见到了我们一直规避的挑战。数字4怎么处理呢？

roman/12/RomanConverterTest.cpp

```
EXPECT_THAT(convert(4), Eq("IV"));
```

我们可以尝试通过引入一点减法逻辑来支持数字4。但这样做似乎给我们的代码带来了一些复杂的逻辑弯路和盘杂的条件分支。唔……

记住，TDD不是无脑练习（参见3.3节）。每一步都需要深思熟虑，包括下一步该是什么样。就目前而言，编写可以驱动开发一个更简单实现的测试行为对我们来说已经非常奏效了，因为它反映了相似的模式。确保代码在每一步都能被正确重构，能够提高我们遵循这一不断简化的路数的能力。

偶尔需要更深刻的洞察力才能达到最佳结果。有时它们来的很容易，有时却很难。编写的程序越多，探寻的东西越多，灵光一现的机会就越多。TDD很棒的一点是，它提供了这些实践机会。尝试更好的代码，你很快就能知道它是否工作。如果不能，那么就回滚掉代码并尝试其他方案。

如果将罗马数字4作为单独的数字，不考虑它需要两个字母（IV）呢？譬如，试想罗马人用单独的特殊字符来表征它。这样的话，它仅仅是转换表中的另一个简单项。

roman/12/RomanConverterTest.cpp

```
const auto arabicToRomanConversions = {
   make_pair(100u, "C"),
   make_pair(50u, "L"),
   make_pair(10u, "X"),
   make_pair(5u, "V"),
➤  make_pair(4u, "IV"),
   make_pair(1u, "I") };
```

整个算法没有变。最后以最终断言收尾，这些断言验证了算法对其他基于减法的数字的支持，包括9、40、90、400和900，还有一些没有添加的数字（500和1000）。对于这些断言，我将convert n to roman作为Google搜索关键字，然后把获得的结果当作断言中的期望值[①]。

roman/13/RomanConverterTest.cpp

```
#include "gmock/gmock.h"

#include <vector>
#include <string>

using namespace ::testing;
using namespace std;

string convert(unsigned int arabic)
{
   const auto arabicToRomanConversions = {
      make_pair(1000u, "M"),
      make_pair(900u, "CM"),
      make_pair(500u, "D"),
      make_pair(400u, "CD"),
      make_pair(100u, "C"),
      make_pair(90u, "XC"),
      make_pair(50u, "L"),
      make_pair(40u, "XL"),
      make_pair(10u, "X"),
      make_pair(9u, "IX"),
      make_pair(5u, "V"),
      make_pair(4u, "IV"),
      make_pair(1u, "I") };

   string roman{""};
```

① n是阿拉伯数字。——译者注

```
   for (auto arabicToRoman: arabicToRomanConversions)
      while (arabic >= arabicToRoman.first)
      {
         roman += arabicToRoman.second;
         arabic -= arabicToRoman.first;
      }
   return roman;
}

TEST(RomanConverter, CanConvertPositiveDigits) {
   EXPECT_THAT(convert(1), Eq("I"));
   EXPECT_THAT(convert(2), Eq("II"));
   EXPECT_THAT(convert(3), Eq("III"));
   EXPECT_THAT(convert(4), Eq("IV"));
   EXPECT_THAT(convert(5), Eq("V"));
   EXPECT_THAT(convert(10), Eq("X"));
   EXPECT_THAT(convert(11), Eq("XI"));
   EXPECT_THAT(convert(12), Eq("XII"));
   EXPECT_THAT(convert(13), Eq("XIII"));
   EXPECT_THAT(convert(20), Eq("XX"));
   EXPECT_THAT(convert(50), Eq("L"));
   EXPECT_THAT(convert(80), Eq("LXXX"));
   EXPECT_THAT(convert(100), Eq("C"));
   EXPECT_THAT(convert(288), Eq("CCLXXXVIII"));
   EXPECT_THAT(convert(2999), Eq("MMCMXCIX"));
   EXPECT_THAT(convert(3447), Eq("MMMCDXLVII"));
   EXPECT_THAT(convert(1513), Eq("MDXIII"));
}
```

大功告成！（除了一个限定，即数字必须在1~4000）。

我们最终得到了一个简短、简单且准确的算法实现。在网上搜索一下就能得到许多其他解决方案，它们远比这复杂，而且没有任何其他多余的优点。遵循TDD并正确重构总是能够得到优化的解决方案。

B.3　熟能生巧

如果你还没有做，那就按照本附录中的步骤完成罗马数字转换器的构建。但不要止步于此，再实现一遍这个转换器，这一次*不要*参考书中的步骤。思考下一步该怎么做、编写一个测试、找一个简单的方法实现代码并重构代码。然后再做一遍——不用立刻做，或许一天或者一周以后。每次做的时候都要记下时间，努力减少敲击键盘的次数和完成整个方案所需要的精力。如果你已经有一段时间不用TDD或C++了，可以以此作为热身训练。

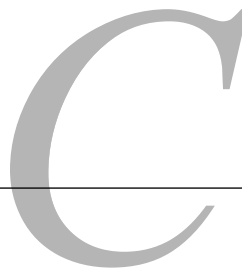
B.4　结束语

本附录中的另一个示例展示了怎样利用TDD增量地开发一个算法。当然，事情并不总是那么

顺利。遇到新问题时，可能会选错出发点，这时需要撤销少量代码，重新尝试。以罗马转换器这类短小的Kata来练习，可以让你对采取短小、增量的方式达成解决方案感到更加舒适。Kata的更多相关细节可以参见11.5节。

附录 C

参考文献



Gojko Adzic,《实例化需求：团队如何交付正确的软件》，张昌贵、张博超译。人民邮电出版社，2013年第一版。

Kent Beck,《测试驱动开发》，孙平平、张小龙、赵辉等译，中国电力出版社，2003出版。

Marcus Gärtner,《验收测试驱动开发：ATDD实例详解》，张绍鹏，冯上译，人民邮电出版社，2013年第一版。

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts.《重构：改善既有代码的设计》，熊节译，人民邮电出版社，2015年第二版。

Michael Feathers,《修改代码的艺术》，侯伯薇译，机械工业出版社，2014年第一版。

Robert C. Martin,《敏捷软件开发（原则模式与实践）》，邓辉译，清华大学出版社，2003年第一版。

Robert C. Martin,《代码整洁之道》，韩磊译，人民邮电出版社，2010年第一版。

[BE12] Daniel Brolund and Ola Ellnestam. *Behead Your Legacy Beast: Refactor and Restructure Relentlessly with the Mikado Method.* Daniel Brolund, Ola Ellnestam, http://www.agical.com,2012.

[Bac12] Emily Bache. *The Coding Dojo Handbook.* LeanPub, https://leanpub.com,2012.

[Bec00] Kent Beck.*Extreme Programming Explained: Embrace Change.*Addison- Wesley Longman, Reading, MA, 2000.

[EB14] Ola Ellnestam and Daniel Brolund. *The Mikado Method.* Manning Publications Co., Greenwich, CT, 2014.

[FP09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests.*Addison-Wesley Longman, Reading, MA, 2009.

[Gre07] James W. Grenning. Embedded Test Driven Development Cycle.*Embedded Systems Conference.*Submissions, 2004, 2006, 2007.

[Gre10] James W. Grenning.*Test Driven Development for Embedded C.* The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2010.

[LO11] Jeff Langr and Tim Ottinger.*Agile in a Flash: Speed-Learning Agile Software Development.*The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2011.

[Lan05] Jeff Langr. *Agile Java: Crafting Code With Test-Driven Development.*Prentice Hall, Englewood Cliffs, NJ, 2005.

[Lan99] Jeff Langr.*Essential Java Style: Patterns for Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1999.

[MFC01] Tim MacKinnon, Steve Freeman, and Philip Craig. Endo-Testing: Unit Testing with Mock Objects.*Extreme Programming Examined.*1:287-302, 2001.

[Mes07] Gerard Meszaros. *xUnit Test Patterns*. Addison-Wesley, Reading, MA, 2007.

[OL11] Tim Ottinger and Jeff Langr. *Agile in a Flash.* The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2011.

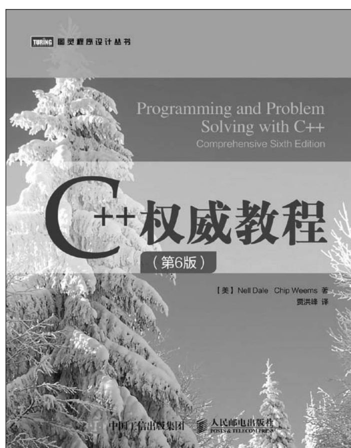
版权声明

延展阅读



▶ 程序员版《人性的弱点》，提升职业生涯软技能

书号：978-7-115-43418-0
定价：45.00 元



▶ C++ 精髓概念与实战案例全解读
▶ 全球数百所高校计算机编程入门教程

书号：978-7-115-40792-4
定价：149.00 元



▶ Martin Fowler 经典著作《重构》的极佳补充
▶ 通过发现和消除设计中的坏味来改善软件质量
▶ IBM 软件工程首席科学家 Grady Booch 作序推荐

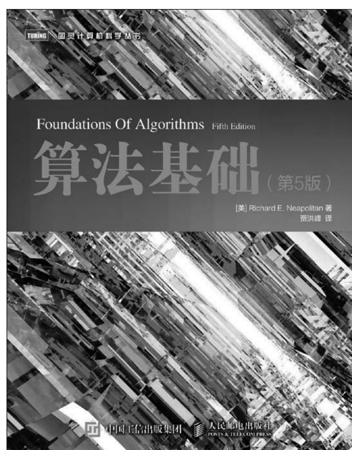
书号：978-7-115-43124-0
定价：59.00 元

延 展 阅 读



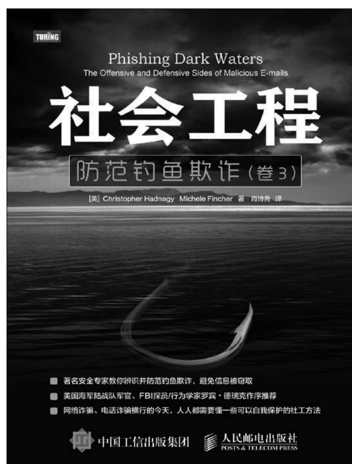
▶ **通过 Netflix、Amazon 等多个业界案例，从微服务架构演进到原理剖析，全面讲解建模、集成、部署等微服务所涉及的各种主题**

书号：978-7-115-42026-8
定价：69.00 元



▶ **海外高校广泛采用的算法教材之一，唯一一本涵盖遗传算法的同类教材**

书号：978-7-115-41657-5
定价：99.00 元



▶ **著名安全专家 Christopher Hadnagy《社会工程》系列之三，教你辨识并防范钓鱼欺诈，避免信息被窃取，全面防护系统安全**

书号：978-7-115-43547-7
定价：49.00 元

测试驱动开发是C++软件开发的一种实践方式，能够有效减少系统中的缺陷，有助于编写更易维护的代码，从而让软件能更从容地应对不断变化的需求。本书是关于现代C++测试驱动开发的综合指南，内容涵盖测试驱动开发的所有相关知识，并提供大量编程实践和软件设计原则，适合所有层次的C++程序员，也可供Java、C#、Python等程序员参考。

- 如何使用测试驱动开发改善遗留的C++系统
- 如何确定和处理令人厌烦的系统依赖性
- 如何在C++中进行棘手的依赖注入
- 如何在C++中使用测试工具辅助测试驱动开发
- 帮助测试驱动开发的C++11新功能

Modern C++ Programming with Test-Driven Development

Code Better, Sleep Better

“这本书中充满了有趣的、编写良好的C++代码，但讨论的内容远不止现代C++编程，还讲述了设计原则、编码实践和技巧……这也是我见过测试驱动开发方面内容最完整、论述最好的书！”

——Bob大叔，敏捷大师，Object Mentor公司创始人，《代码整洁之道》作者

“Jeff Langr又写了一本很棒的书。这一次他把测试驱动开发引入到C++世界。Jeff的示例让我们近距离地领略到好的测试驱动开发方法简约的一面。他解释了为什么要以这种方式工作，然后提出了重要的实践细节，涉及测试替身、与遗留代码打交道的方法、对付多线程代码，还有很多其他的内容。每个使用C++的开发者都应该拥有这本书。”

——Ron Jeffries，极限编程方法学创始人之一，《软件开发本质论》作者

The Pragmatic Programmers

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-43895-9

定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks